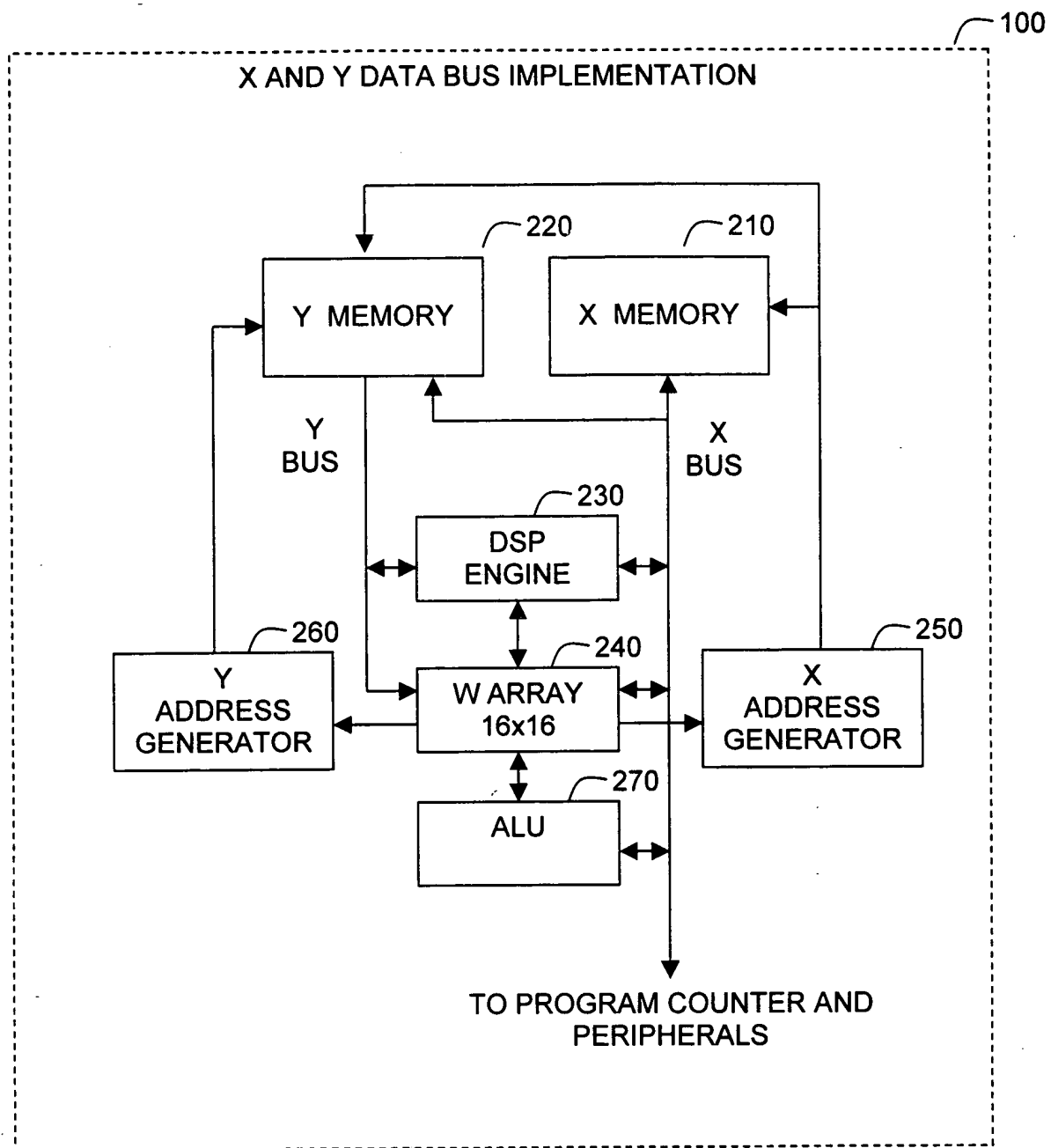
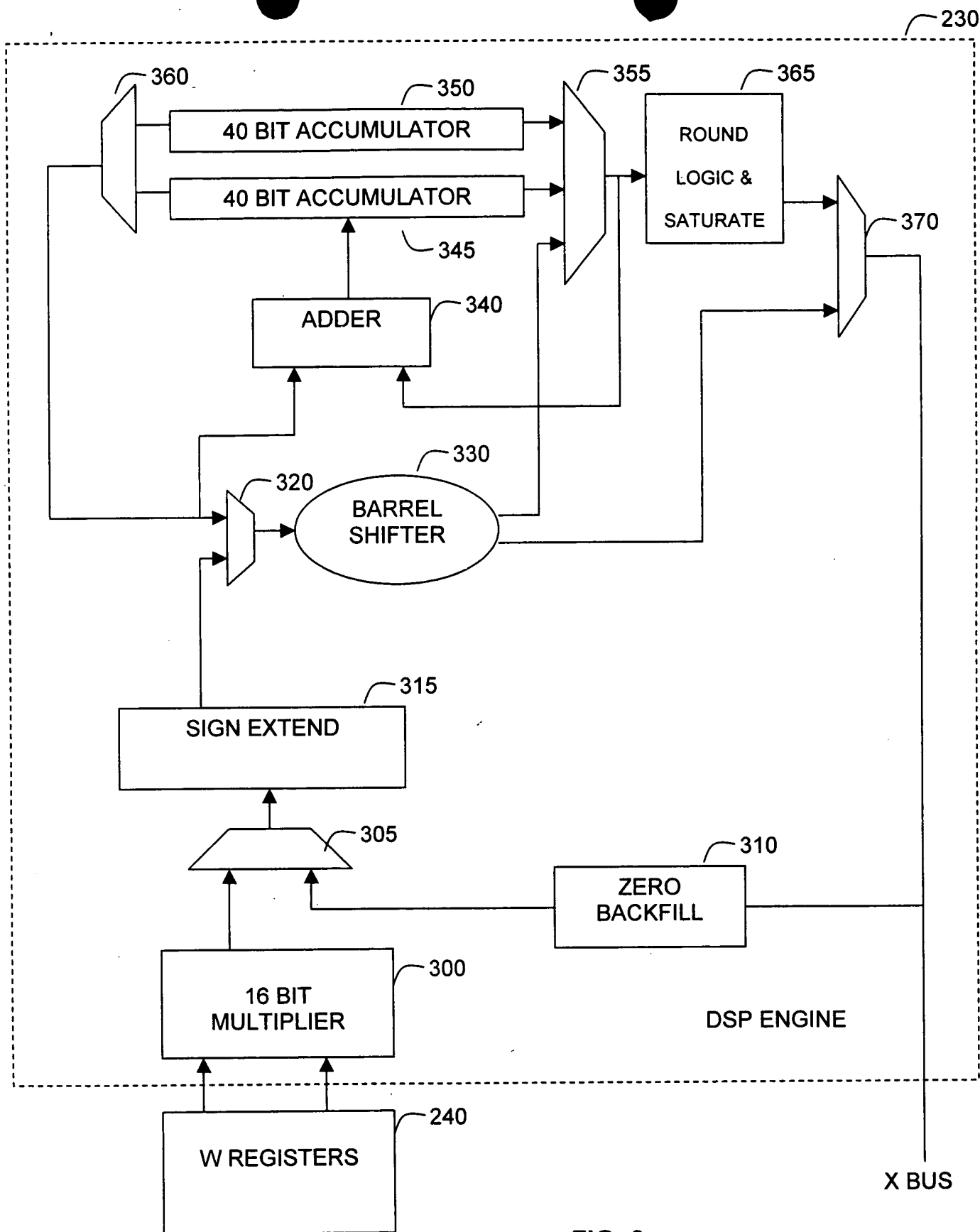


**FIG. 1**

09870457-060101



**FIG. 2**



**FIG. 3**

FIG. 4A

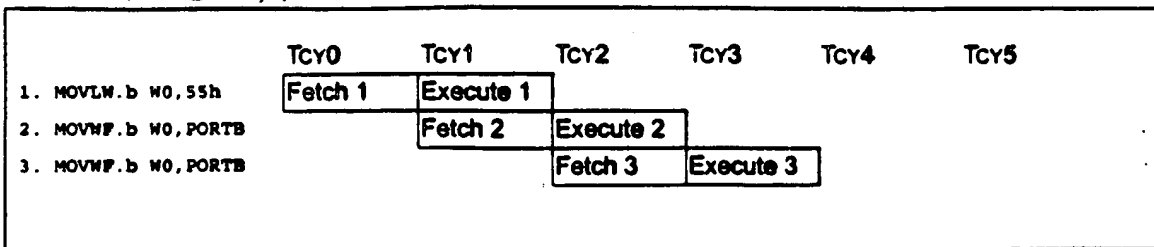


FIG. 4B

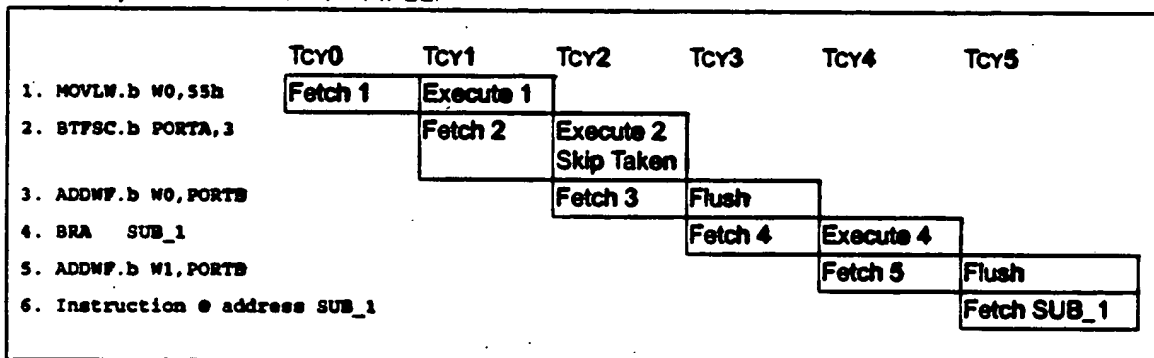




FIG. 4C

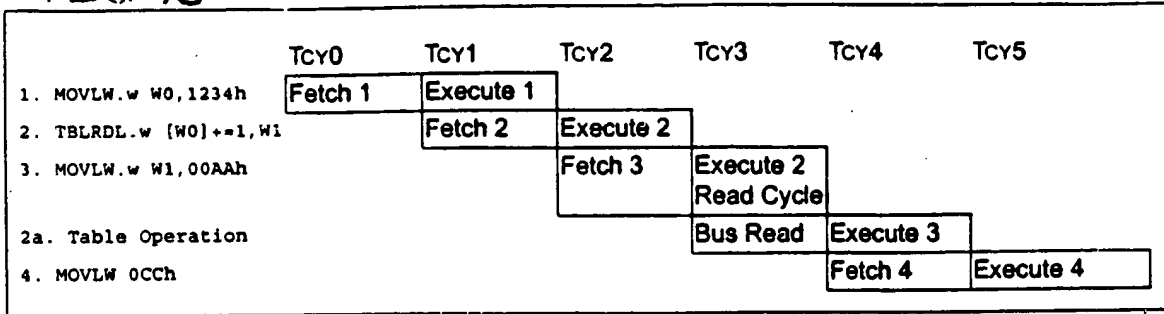


FIG. 4D

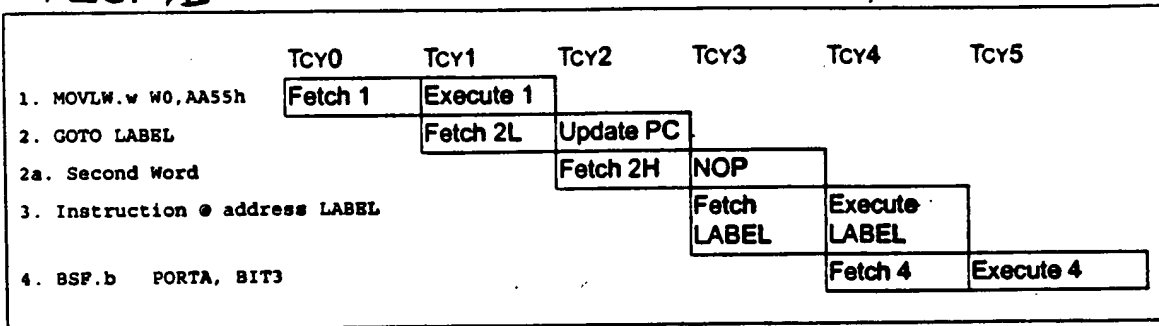
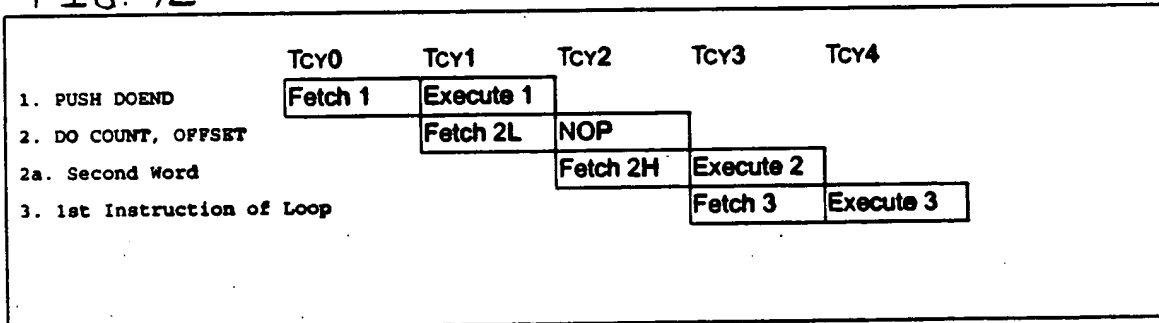


FIG. 4E



# dsPIC Core DOS

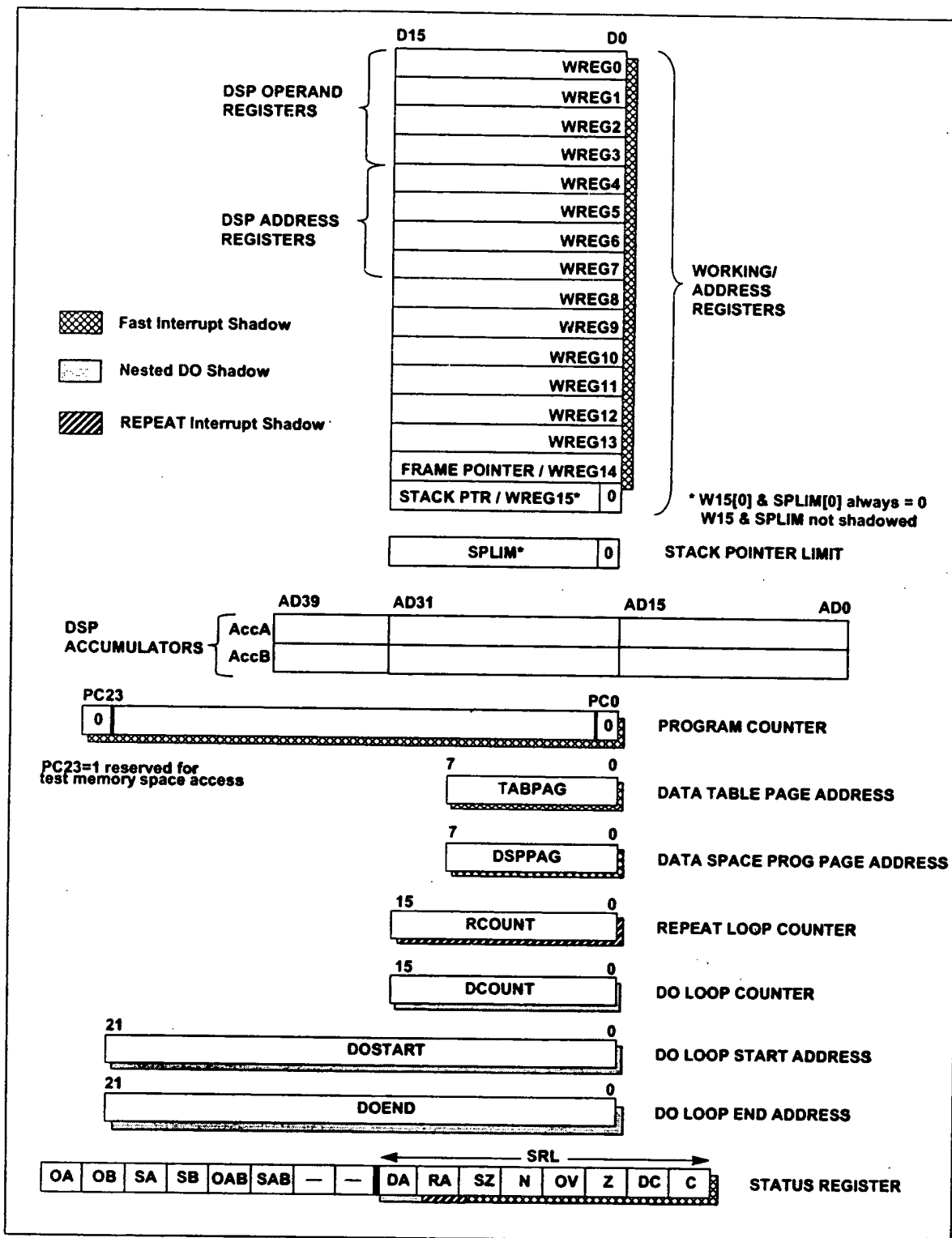


FIGURE 1-33: PROGRAMMERS MODEL

THIS DOCUMENT IS UNCONTROLLED UNLESS OTHERWISE STAMPED. It is the users responsibility to ensure this is the latest revision prior to using or referencing this document.	Page	SPEC. NO.	REV.
2000/12/08 Microchip Technology, Inc. CONFIDENTIAL AND PROPRIETARY	36 of 167	DOS-00200	A Draft

FIG 5

UNITED STATES PATENT AND TRADEMARK OFFICE  
DOCUMENT CLASSIFICATION BARCODE SHEET



**Miscellaneous**

**10**

09870457.060101

APPENDIX A

09870457 060101

Improper Appendix.

Appendix is limited to only  
program listings

[illegible]

Certain W registers have implied utilization in the instruction set. W0-W3 are used as the operands for DSP instructions. W4-W7 are used as the prefetch addresses for DSP instructions. W14 is the frame pointer utilized by the LNK and ULNK instructions. W15 acts as the stack pointer.

Register	
W0	MAC operand; Default Ww
W1	MAC operand
W2	MAC operand; MULWF product LSB
W3	MAC operand; MULWF product MSB
W4	MAC prefetch address
W5	MAC prefetch address
W6	MAC prefetch address
W7	MAC prefetch address
W8	MAC prefetch offset
W9	MAC write back address
W10	
W11	
W12	
W13	
W14	Frame Pointer
W15	Stack Pointer

W0 serves as the default Ww register for file register instructions. In this capacity, Ww acts as the W register in C16 and C18 compatible instructions.

When a byte is moved into a W register, the byte is written into the LSbyte of the register and the MSbyte is left alone. Byte operations on the registers will operate on the LSbyte of the register. The MSbyte of the register is left alone. For byte operations, the status flags will be adjusted to respond to the <7:0> bits of the register. For example, the carry bit will originate from ALU<7>. When a byte is moved from a W register, the source is the LSbyte and it overwrites the target byte in the memory. Other bytes are not affected.

The Bit operation instructions that use the W registers can address bytes or words without the requirement for a B bit.

This works by making the bit field selection look at the LSB of the word or byte being addressed by the W register.

$$W_0 = 1000$$

**BCLR W0,#5** : Clear 5th bit in word 1000

**BCLR W0,#13 ; Clear 13th bit in word 1000**

**BCLR W1,#5 ; Clear 5th bit in byte 1001, same as  
clear 13th bit in word 1000.**

BCLR W1,#13 ; Invalid, same as

clear 13th bit in word 1000.

## 5.4 Using 10-bit literals

The instructions that have 10-bit literals have byte and word modes. For byte instructions, the literal is truncated at 8 bits. If the user specifies a signed value {-128... -1}, the truncated 2's compliment is coded. Unsigned values may range from {0 ... 255}. For word instructions, the literal is sign extended to 16-bits.

**TABLE 5-2: 10-BIT LITERAL CODING**

Literal Value	If B=0 (Word)	If B=1 (Byte)
	kk kkkk kkkk	kk kkkk kkkk
-512	10 0000 0000	n/a
-511	10 0000 0001	n/a
-129	11 0111 1111	n/a
-128	11 1000 0000	11 1000 0000
-2	11 1111 1110	11 1111 1110
-1	11 1111 1111	11 1111 1111
0	00 0000 0000	00 0000 0000
1	00 0000 0001	00 0000 0001
2	00 0000 0010	00 0000 0010
127	00 0111 1111	00 0111 1111
128	00 1000 0000	00 1000 0000
255	00 1111 1111	00 1111 1111
256	01 0000 0000	n/a
511	11 1111 1111	n/a

5.5 Program Memory Addressing

Program memory contains a user space and a test space. The most significant bit (PMA<23>) of the program memory address selects user / test space. The least significant bit (PMA<0>) selects a byte for data addressing and table addressing modes.

Program memory addresses coded into instructions are coded in a lit23 or Slit16 format.

The lit23 format encodes a direct address that represents PMA<22:0>. PMA<23> is not valid user space and is not encoded.

The Slit16 format encodes an instruction count offset. The offset is added to the PC to generate the next address. The Slit16 format does not encode the PMA<0> bit as it represents an instruction count. The Slit16<15> bit is sign extended when added to the PC.

FIGURE 5-1: PROGRAM MEMORY ADDRESSING

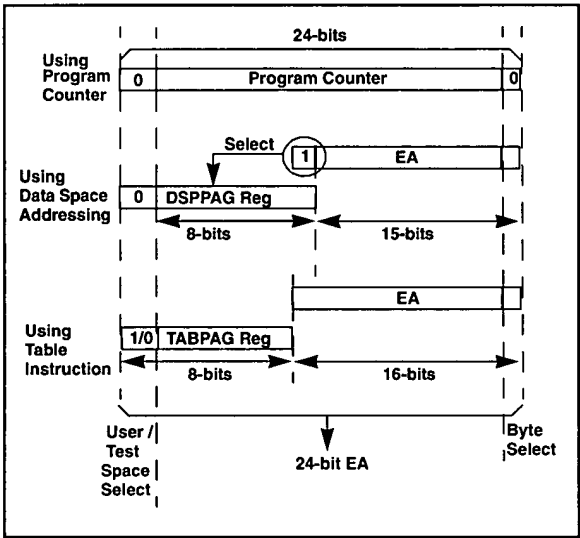


FIGURE 5-2: "CALL lit23" MAP TO PC

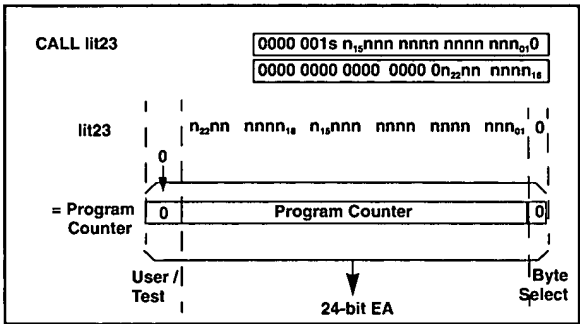


FIGURE 5-3: "BRA Slit16" MAP TO PC

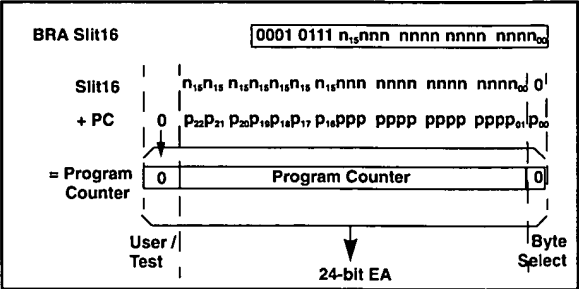


FIGURE 5-4: "GOTO Wn" MAP TO PC

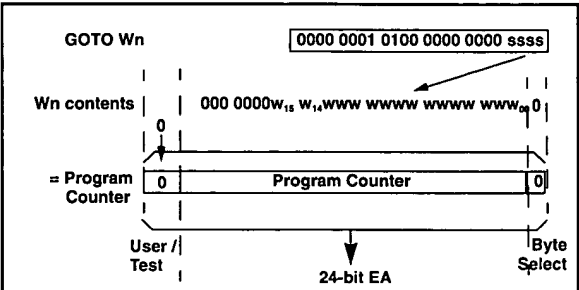
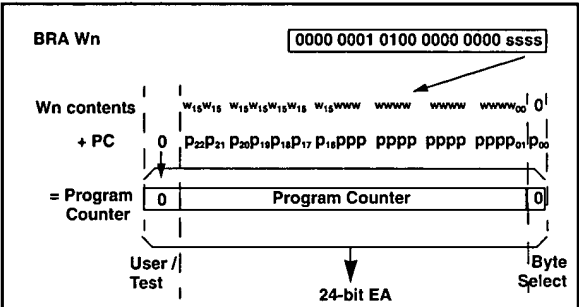


FIGURE 5-5: "BRA Wn" MAP TO PC



5.6 Shadows

Shadow registers are 1 level deep mini-stack registers attached to several key user registers. A PUSH.S will copy the user registers to the shadows and a POP.S will copy the shadows back to the user registers.

Shadow registers are attached to W0...W15, the STATUS register, and the LCR,LSR,LER registers used by DO and REPEAT instructions.

## 5.7 MAC

The MAC instruction is a pipelined instruction. The first pipeline stage generates the effective addresses of the X and Y data and fetches the X and Y data. The second pipeline stage computes the multiply and accumulate, storing the results into the accumulator.

### 5.7.1 FORMS

The MAC instruction, and variants, can have several formats. Fundamentally, it must specify a target accumulator and a multiplicand and multiplier ( $ACC=X*Y$ ). For Example:

```
MAC A,W0*W1
```

The MAC can also specify a prefetch for the next X or Y operand. The assembler can discriminate the X or Y prefetch based on the register used as the indirect address. [W4] or [W5] specifies the X prefetch and [W6] or [W7] specifies the Y prefetch. If a prefetch is specified, it must have a prefetch destination register. Legal forms of prefetch include:

```
MAC A,W0*W1,W0,[W4] ;X only
```

```
MAC A,W0*W1,W1,[W6] ;Y only
```

```
MAC A,W0*W1,W0,[W4],W1,[W6] ;X,Y
```

A write back can be specified. The write back uses the W9 register as the destination address. In this way, the assembler can discern the write back option.

```
MAC A,W0*W1,[W9] ;WBack only
```

```
MAC A,W0*W1,W0,[W6],W9 ;Y,WBack
```

```
MAC A,W0*W1,W0,[W4],[W9] ;X,Wback
```

```
MAC A,W0*W1,W0,[W4],W1,[W6],W9
```

### 5.7.2 SQUARING OPERATIONS

Squaring in the DSP engine is done with the square PLA opcodes. These are variants of the MAC and MPY opcodes.

For Example:

```
MAC B,W0*W0,W0,[W4],W1,[W6] +=2,W9
```

This instruction will multiply W0 time W0 and write the result in ACCB while doing the prefetch and write back.

The assembler can tell that a MAC or MPY should translate to SQRAC or SQR instructions by finding the  $Wm*Wm$  format.



5.8 File Registers

File registers include parts of user RAM area and the Special Function Registers (SFR). The file register space is 8192 bytes. The file registers are directly addressable using the f field in the file register instructions.

All data addresses are byte addresses. When using byte instructions, the bytes are addressed directly. When using word instructions, the address must be word aligned. The least significant address bit must be 0.

FIGURE 5-6: Data Alignment in Memory

0 0 0 0 0 0 0 0	d <sub>07</sub> d <sub>06</sub> d <sub>05</sub> d <sub>04</sub> d <sub>03</sub> d <sub>02</sub> d <sub>01</sub> d <sub>00</sub>	Byte @ 0x0100
d <sub>07</sub> d <sub>06</sub> d <sub>05</sub> d <sub>04</sub> d <sub>03</sub> d <sub>02</sub> d <sub>01</sub> d <sub>00</sub>	0 0 0 0 0 0 0 0	Byte @ 0x0103
d <sub>15</sub> d <sub>14</sub> d <sub>13</sub> d <sub>12</sub> d <sub>11</sub> d <sub>10</sub> d <sub>09</sub> d <sub>08</sub>	d <sub>07</sub> d <sub>06</sub> d <sub>05</sub> d <sub>04</sub> d <sub>03</sub> d <sub>02</sub> d <sub>01</sub> d <sub>00</sub>	Word @ 0x0104
d <sub>15</sub> d <sub>14</sub> d <sub>13</sub> d <sub>12</sub> d <sub>11</sub> d <sub>10</sub> d <sub>09</sub> d <sub>08</sub>	d <sub>07</sub> d <sub>06</sub> d <sub>05</sub> d <sub>04</sub> d <sub>03</sub> d <sub>02</sub> d <sub>01</sub> d <sub>00</sub>	DWord @ 0x0106
d <sub>31</sub> d <sub>30</sub> d <sub>29</sub> d <sub>28</sub> d <sub>27</sub> d <sub>26</sub> d <sub>25</sub> d <sub>24</sub>	d <sub>23</sub> d <sub>22</sub> d <sub>21</sub> d <sub>20</sub> d <sub>19</sub> d <sub>18</sub> d <sub>17</sub> d <sub>16</sub>	QWord @ 0x010A
d <sub>15</sub> d <sub>14</sub> d <sub>13</sub> d <sub>12</sub> d <sub>11</sub> d <sub>10</sub> d <sub>09</sub> d <sub>08</sub>	d <sub>07</sub> d <sub>06</sub> d <sub>05</sub> d <sub>04</sub> d <sub>03</sub> d <sub>02</sub> d <sub>01</sub> d <sub>00</sub>	
d <sub>31</sub> d <sub>30</sub> d <sub>29</sub> d <sub>28</sub> d <sub>27</sub> d <sub>26</sub> d <sub>25</sub> d <sub>24</sub>	d <sub>23</sub> d <sub>22</sub> d <sub>21</sub> d <sub>20</sub> d <sub>19</sub> d <sub>18</sub> d <sub>17</sub> d <sub>16</sub>	
d <sub>47</sub> d <sub>46</sub> d <sub>45</sub> d <sub>44</sub> d <sub>43</sub> d <sub>42</sub> d <sub>41</sub> d <sub>40</sub>	d <sub>39</sub> d <sub>38</sub> d <sub>37</sub> d <sub>36</sub> d <sub>35</sub> d <sub>34</sub> d <sub>33</sub> d <sub>32</sub>	
d <sub>63</sub> d <sub>62</sub> d <sub>61</sub> d <sub>60</sub> d <sub>59</sub> d <sub>58</sub> d <sub>57</sub> d <sub>56</sub>	d <sub>55</sub> d <sub>54</sub> d <sub>53</sub> d <sub>52</sub> d <sub>51</sub> d <sub>50</sub> d <sub>49</sub> d <sub>48</sub>	Byte @ 0x0112
0 0 0 0 0 0 0 0	d <sub>07</sub> d <sub>06</sub> d <sub>05</sub> d <sub>04</sub> d <sub>03</sub> d <sub>02</sub> d <sub>01</sub> d <sub>00</sub>	

5.9 Carry and Borrow in PIC instructions

The PIC uses one unified carry and borrow bit, the C bit in the status register. The following examples show the functionality of the carry / borrow.

If a normal add generates a carry out of the 15th bit, the carry bit is set.

```
ADD 1 + 65535
  1 = 0000 0000 0000 0001
+ 65535 = 1111 1111 1111 1111
-----
  0 = 0000 0000 0000 0000
  C = 1
  Z = 1
  N = 0
  OV = 0
```

An add carry will use the carry bit as an additional input. If the add generates a carry out of the 15th bit, the carry bit is set.

```
ADDC 1 + 65535, no carry in
  1 = 0000 0000 0000 0001
+ 65535 = 1111 1111 1111 1111
  C = 0
-----
  0 = 0000 0000 0000 0000
  C = 1
  Z = 1
  N = 0
  OV = 0
```

```
ADDC 1 + 65535, carry in
  1 = 0000 0000 0000 0001
+ 65535 = 1111 1111 1111 1111
  C = 1
-----
  0 = 0000 0000 0000 0001
  C = 1
  Z = 0
  N = 0
  OV = 0
```

A subtract instruction inverts the bits of the subtrahend, forces the carry in to 1 and does an add. This has the effect of generating the 2's complement of the subtrahend. If the add generates a carry out of the 15th bit, the carry bit is set. However, in the case of a subtract, the carry bit is viewed as a BORROW bit. So a 1 in the carry bit indicates no borrow. A 0 in the carry bit indicates a borrow.

Subtracting 3 - 2 generates no borrow, so the C bit is 1.

```
SUB 3 - 2
  3 = 0000 0000 0000 0011
+ not 2 = 1111 1111 1111 1101
  C = 1
-----
  1 = 0000 0000 0000 0001
  C = 1
  Z = 0
  N = 0
  OV = 0
```

Subtracting 3 - 3 generates no borrow, so the C bit is 1. The Z bit indicates a zero result.

```
SUB 3 - 3
  3 = 0000 0000 0000 0011
+ not 3 = 1111 1111 1111 1100
  C = 1
-----
  0 = 0000 0000 0000 0000
  C = 1
  Z = 1
  N = 0
  OV = 0
```

Subtracting 2 - 3 generates a borrow, so the C bit is 0. The N bit indicates a negative result.

```
SUB 2 - 3
  2 = 0000 0000 0000 0010
+ not 3 = 1111 1111 1111 1100
  C = 1
-----
 -1 = 1111 1111 1111 1111
  C = 0
  Z = 0
  N = 1
  OV = 0
```

A subtract with borrow instruction inverts the bits of the subtrahend, leaves the carry at its previous state and does an add. This has the effect of generating the 2's complement of the subtrahend while inputting a BORROW bit.

Subtract / borrow 3 - 2 with no borrow in generates no borrow, so the C bit is 1.

```
SUBB 3 - 2, no borrow in
  3 = 0000 0000 0000 0011
+ not 2 = 1111 1111 1111 1101
  C = 1
-----
  1 = 0000 0000 0000 0001
  C = 1
  Z = 0
  N = 0
  OV = 0
```

Subtract / borrow 3 - 2 with borrow in generates no borrow, so the C bit is 1. The result is 0, so the Z bit is set.

```
SUBB 3 - 2, borrow in
  3 = 0000 0000 0000 0011
+ not 2 = 1111 1111 1111 1101
  C = 0
-----
  0 = 0000 0000 0000 0000
  C = 1
  Z = 1
  N = 0
  OV = 0
```

Subtract / borrow 2 - 3 with borrow in generates a borrow, so the C bit is 0. The N bit indicates a negative result.

```
SUBB 2 - 3, borrow in
  2 = 0000 0000 0000 0010
+ not 3 = 1111 1111 1111 1100
  C = 0
-----
 -2 = 1111 1111 1111 1110
  C = 0
  Z = 0
  N = 1
  OV = 0
```

	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431	2432	2
--	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	---

```

SUB 32760 - -32768
      32760 = 0111 1111 1111 1000
+ not 32768 = 0111 1111 1111 1111
      C = 1
-----
      -8 = 1111 1111 1111 1000
      C = 0
      Z = 0
      N = 1
      OV = 1

```

## 5.11 Branch Conditions

### TABLE 5-3: BRANCH CONDITIONS

Instruction	Status Test
BRA C,Slit16	C
BRA GE,Slit16	$(\overline{N} \& \overline{OV}) \parallel (N \& OV)$
BRA GEU,Slit16	C
BRA GT,Slit16	$(\overline{Z} \& \overline{N} \& \overline{OV}) \parallel (\overline{Z} \& N \& OV)$
BRA GTU,Slit16	$C \& \overline{Z}$
BRA LE,Slit16	$Z \parallel (\overline{N} \& OV) \parallel (N \& \overline{OV})$
BRA LEU,Slit16	$\overline{C} \parallel Z$
BRA LT,Slit16	$(\overline{N} \& OV) \parallel (N \& \overline{OV})$
BRA LTU,Slit16	$\overline{C}$
BRA N,Slit16	N
BRA NC,Slit16	$\overline{C}$
BRA NN,Slit16	$\overline{N}$
BRA NOV,Slit16	$\overline{OV}$
BRA NZ,Slit16	$\overline{Z}$
BRA OV,Slit16	OV
BRA Z,Slit16	Z

Minu	Subtr	C	Z	N	OV	LT	LTU	LE	LEU	GE	GEU	GT	GTU
3	2	1	0	0	0	0	0	0	0	1	1	1	1
3	3	1	1	0	0	0	0	1	1	1	1	0	0
2	3	0	0	1	0	1	1	1	1	0	0	0	0
32760 -or- 32760	-32768  32768	0	0	1	1	0	1	0	1	1	0	1	0
-32760 -or- 32776	32767  32767	1	0	0	1	1	0	1	0	0	1	0	1

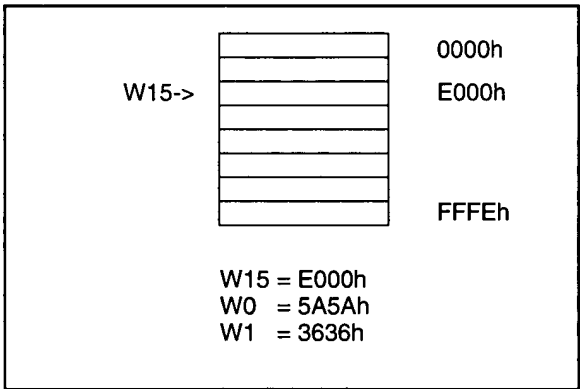
5.12 Stack operation

The dsPIC stack is a software stack implemented in user RAM area. While the device has provisions to allow pointer manipulation on any of the 16 W registers, W15 is the assumed stack pointer.

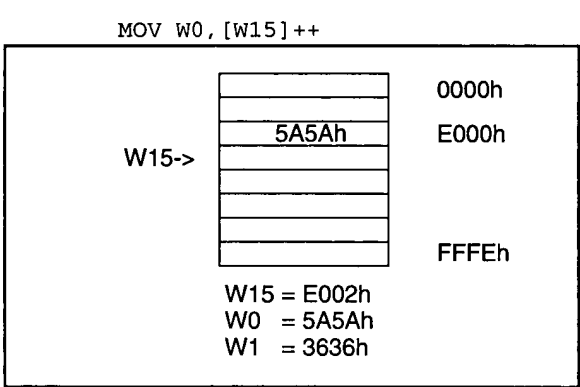
The stack starts at lower memory and grows towards high memory. The stack pointer points to the next available location. The stack pointer is manipulated with the source and destination addressing modes as shown in Table 1-7 and Table 1-8.

A push is MOV W0, [W15]++.  
A pop is MOV [W15--], W0 .

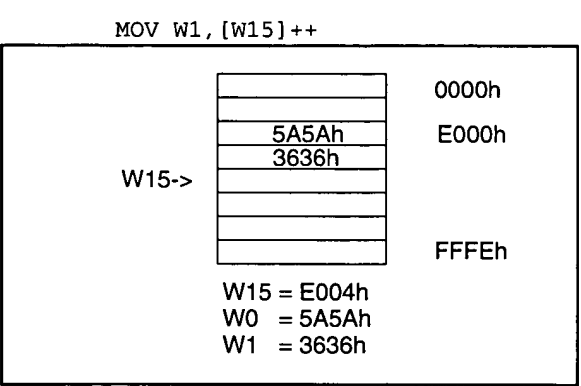
Stack Pointer at Initialization



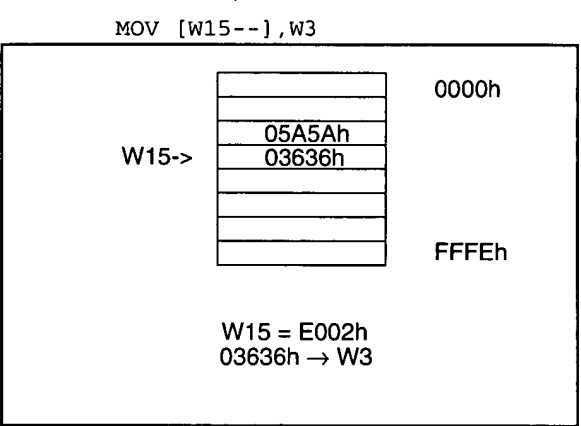
Stack Pointer after Push



Stack Pointer after Push



Stack Pointer after Pop



5.13 Multi-word Move operations

The multi-word move instructions manipulated with the source and destination addressing modes as shown in Table 1-7 and Table 1-8.

FIGURE 5-7: MOV.D OPERATION

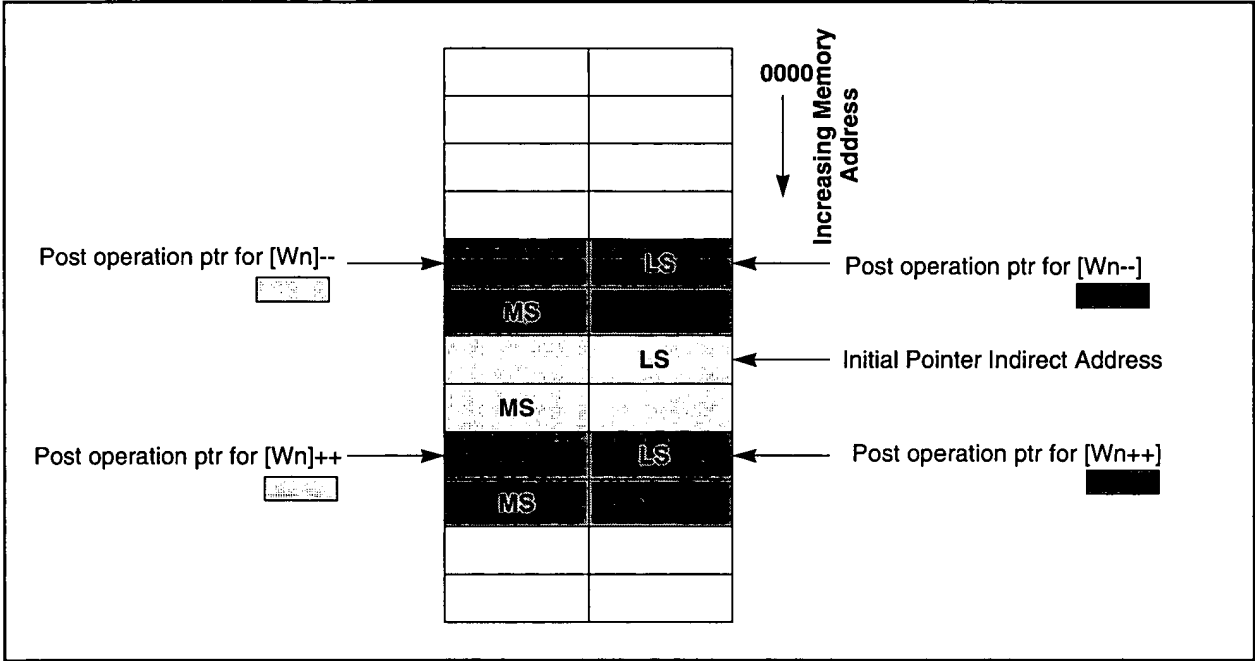


TABLE 5-5: STDW OPERATION

Instr. Cycle	Ws	[Ws]	[Ws++]	[Ws--]	[Ws]++	[Ws]--
Q1 Q2	W(nd)→Ws	Ws=Ws+2 W(nd+1)→(Ws)	Ws=Ws+6 W(nd+1)→(Ws)	Ws=Ws-2 W(nd+1)→(Ws)	W(nd)→(Ws) Ws=Ws+2	W(nd)→(Ws) Ws=Ws+2
Q3 Q4	W(nd+1)→W(s+1)	Ws=Ws-2 W(nd)→(Ws)	Ws=Ws-2 W(nd)→(Ws)	Ws=Ws-2 W(nd)→(Ws)	W(nd+1)→(Ws) Ws=Ws+2	W(nd+1)→(Ws) Ws=Ws-6

TABLE 5-6: LDDW OPERATION

Instr. Cycle	Ws	[Ws]	[Ws++]	[Ws--]	[Ws]++	[Ws]--
Q1 Q2	Ws→W(nd)	Ws=Ws+2 (Ws)→W(nd+1)	Ws=Ws+6 (Ws)→W(nd+1)	Ws=Ws-2 (Ws)→W(nd+1)	(Ws)→W(nd) Ws=Ws+2	(Ws)→W(nd) Ws=Ws+2
Q3 Q4	W(s+1)→W(nd+1)	Ws=Ws-2 (Ws)→W(nd)	Ws=Ws-2 (Ws)→W(nd)	Ws=Ws-2 (Ws)→W(nd)	(Ws)→W(nd+1) Ws=Ws+2	(Ws)→W(nd+1) Ws=Ws-6

FIGURE 5-8: MOV.Q OPERATION

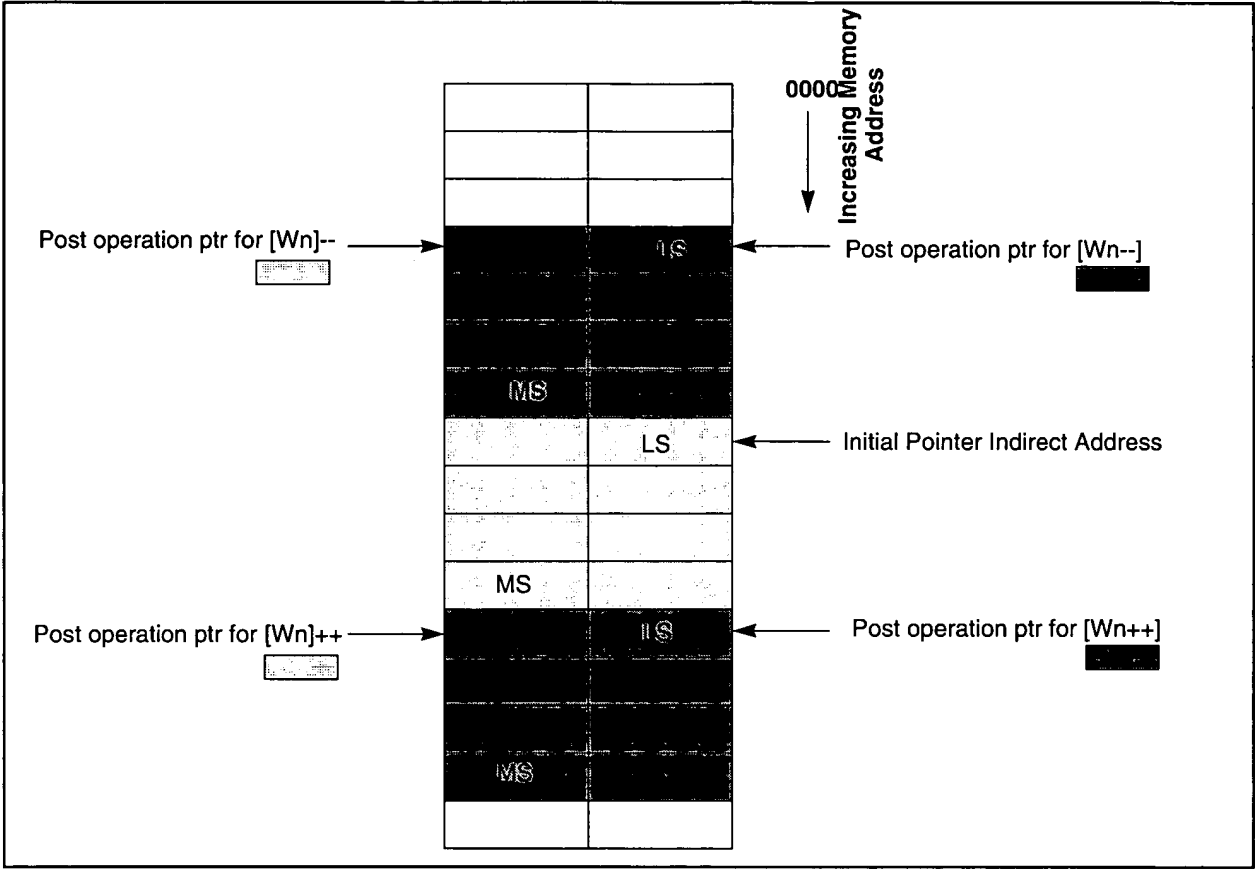


TABLE 5-7: STQW OPERATION

Instr. Cycle	Ws	[Ws]	[Ws++]	[Ws--]	[Ws]++	[Ws]--
Q1 Q2	W(nd)→Ws	Ws=Ws+6 W(nd+3)→(Ws)	Ws=Ws+14 W(nd+3)→(Ws)	Ws=Ws-2 W(nd+3)→(Ws)	W(nd)→(Ws) Ws=Ws+2	W(nd)→(Ws) Ws=Ws+2
Q3 Q4	W(nd+1)→W(s+1)	Ws=Ws-2 W(nd+2)→(Ws)	Ws=Ws-2 W(nd+2)→(Ws)	Ws=Ws-2 W(nd+2)→(Ws)	W(nd+1)→(Ws) Ws=Ws+2	W(nd+1)→(Ws) Ws=Ws+2
Q1 Q2	W(nd+2)→W(s+2)	Ws=Ws-2 W(nd+1)→(Ws)	Ws=Ws-2 W(nd+1)→(Ws)	Ws=Ws-2 W(nd+1)→(Ws)	W(nd+2)→(Ws) Ws=Ws+2	W(nd+2)→(Ws) Ws=Ws+2
Q3 Q4	W(nd+3)→W(s+3)	Ws=Ws-2 W(nd)→(Ws)	Ws=Ws-2 W(nd)→(Ws)	Ws=Ws-2 W(nd)→(Ws)	W(nd+3)→(Ws) Ws=Ws+2	W(nd+3)→(Ws) Ws=Ws-14

**TABLE 5-8: LDQW OPERATION**

Instr. Cycle	Ws	[Ws]	[Ws++]	[Ws--]	[Ws]++	[Ws]--
Q1 Q2	Ws→W(nd)	Ws=Ws+6 W(s+3)→W(nd+3)	Ws=Ws+14 W(s+3)→W(nd+3)	Ws=Ws-2 W(s+3)→W(nd+3)	W(nd)→(Ws) Ws=Ws+2	W(nd)→(Ws) Ws=Ws+2
Q3 Q4	W(s+1)→W(nd+1)	Ws=Ws-2 W(s+2)→W(nd+2)	Ws=Ws-2 W(s+2)→W(nd+2)	Ws=Ws-2 W(s+2)→W(nd+2)	W(s+1)→W(nd+1) Ws=Ws+2	W(s+1)→W(nd+1) Ws=Ws+2
Q1 Q2	W(s+2)→W(nd+2)	Ws=Ws-2 W(s+1)→W(nd+1)	Ws=Ws-2 W(s+1)→W(nd+1)	Ws=Ws-2 W(s+1)→W(nd+1)	W(s+2)→W(nd+2) Ws=Ws+2	W(s+2)→W(nd+2) Ws=Ws+2
Q3 Q4	W(s+3)→W(nd+3)	Ws=Ws-2 Ws→W(nd)	Ws=Ws-2 Ws→W(nd)	Ws=Ws-2 Ws→W(nd)	W(s+3)→W(nd+3) Ws=Ws+2	W(s+3)→W(nd+3) Ws=Ws-14

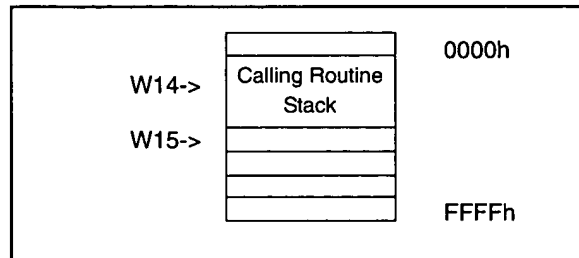
FOR 090 25102360

## 5.14 Link and Unlink Instructions

The link and unlink instructions assume that W15 is a stack pointer and W14 is a frame pointer.

The link instruction is used during a calling sequence.

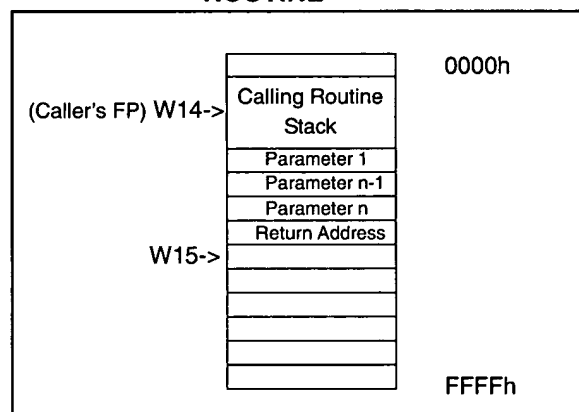
**FIGURE 5-9: STACK AT BEGINNING OF CALLING SEQUENCE**



Before calling the subroutine, the parameters of the routine are pushed on the stack.

```
PUSH W0      ;Push parameter 1
PUSH W1      ;Push parameter n-1
PUSH W2      ;Push parameter n
CALL SUBR
```

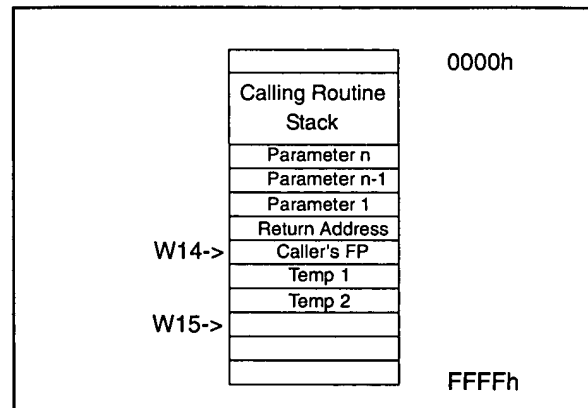
**FIGURE 5-10: STACK AT ENTRY TO ROUTINE**



```
SUBR: LNK 2 ;Allocate 2 words
```

The LNK instruction will push the calling routines FP onto the stack. The new FP will be set to point to the current stack pointer. Then the literal is subtracted from the stack pointer which reserves the amount of memory allocated.

**FIGURE 5-11: STACK AFTER LNK INSTRUCTION**



Inside of the routine, the stack is used to save values. [W14+n] will access the Temp locations used by the routine. [W14-n] is used to access the parameters.

At the end of the routine, the ULNK instruction will copy the FP to the stack pointer then POP the callers FP back to the FP.

```
ULNK ;De-allocate frame
```

This returns the stack back to the state in Figure 5-10.

A return instruction will return to the caller. The caller is responsible for removing the parameters from the stack.

```
RETURN
POP W2 ;Unload parameter 1
POP W1 ;Unload parameter n-1
POP W0 ;Unload parameter n
```

This returns the stack back to the state in Figure 5-9.



## 5.15 Multi-word Shift Instructions

The multi word shift instructions rely on additional special registers. The CARRY1 and CARRY0 registers hold the temporary values of the shift.

### 5.15.1 32-BIT LEFT SHIFTS

The multi-word left shift instructions utilize the shifter associated with the ACCn registers. The instruction can shift 0 to 31 positions. Although the shifter can only implement shifts of up to 15 positions to the left,

by rearranging the storing into the destination registers an apparent shift of 31 positions may be obtained. Figure 5-12 provides an example where the shift amount is 15 or less. The Wnd destination register is aligned with the source and the CARRY0 register contains the shift out results. The CARRY1 register is unused and remains cleared. When the next 16-bit word is shifted, the results are OR'ed with the contents of the CARRY0 register, providing the shift in from the previous shift. The SLMK instruction may be repeated for each 16-bit segment of the multi-word shift.

**FIGURE 5-12: Multi-Word Left Shift by 4 Instruction Execution**

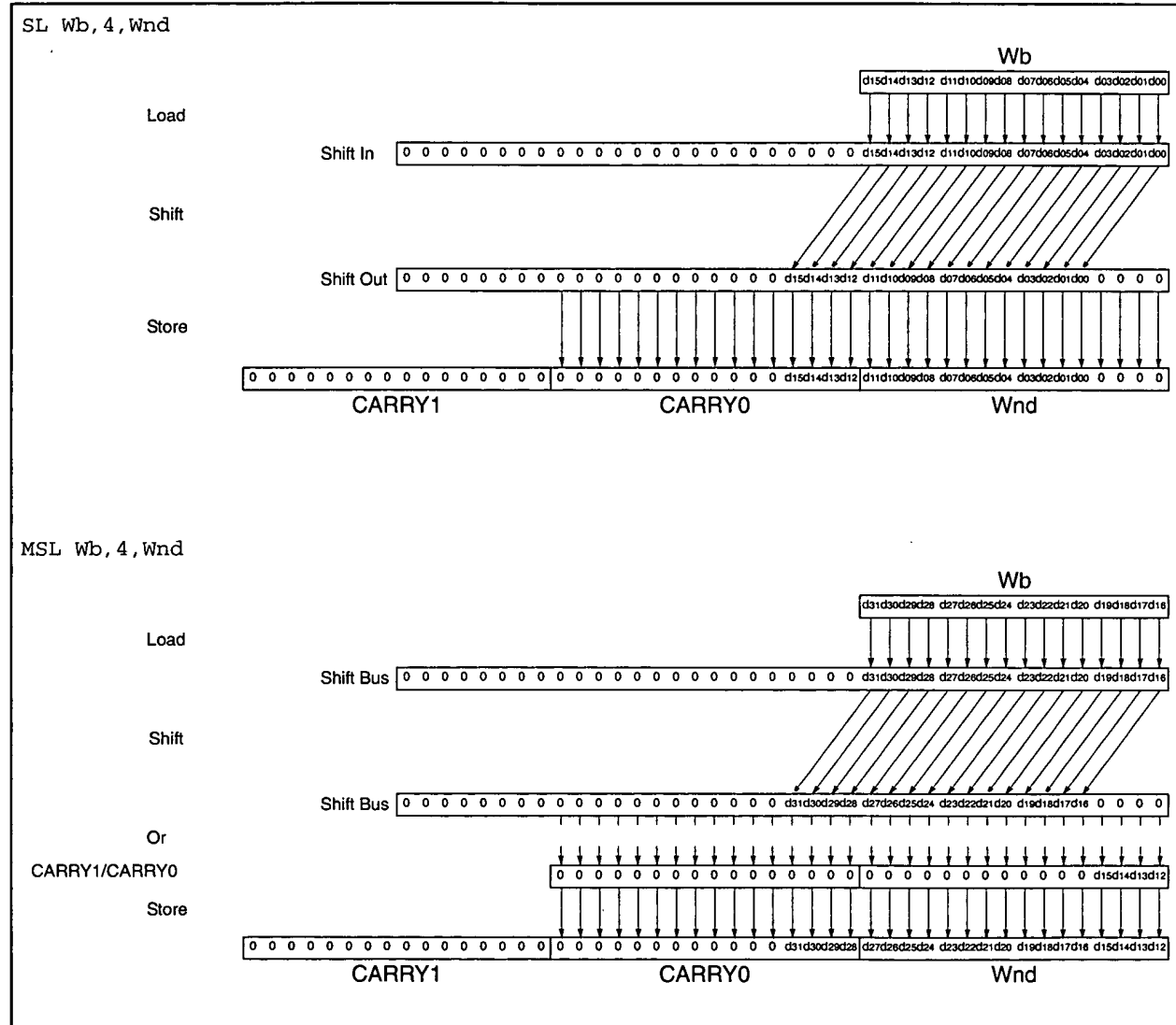
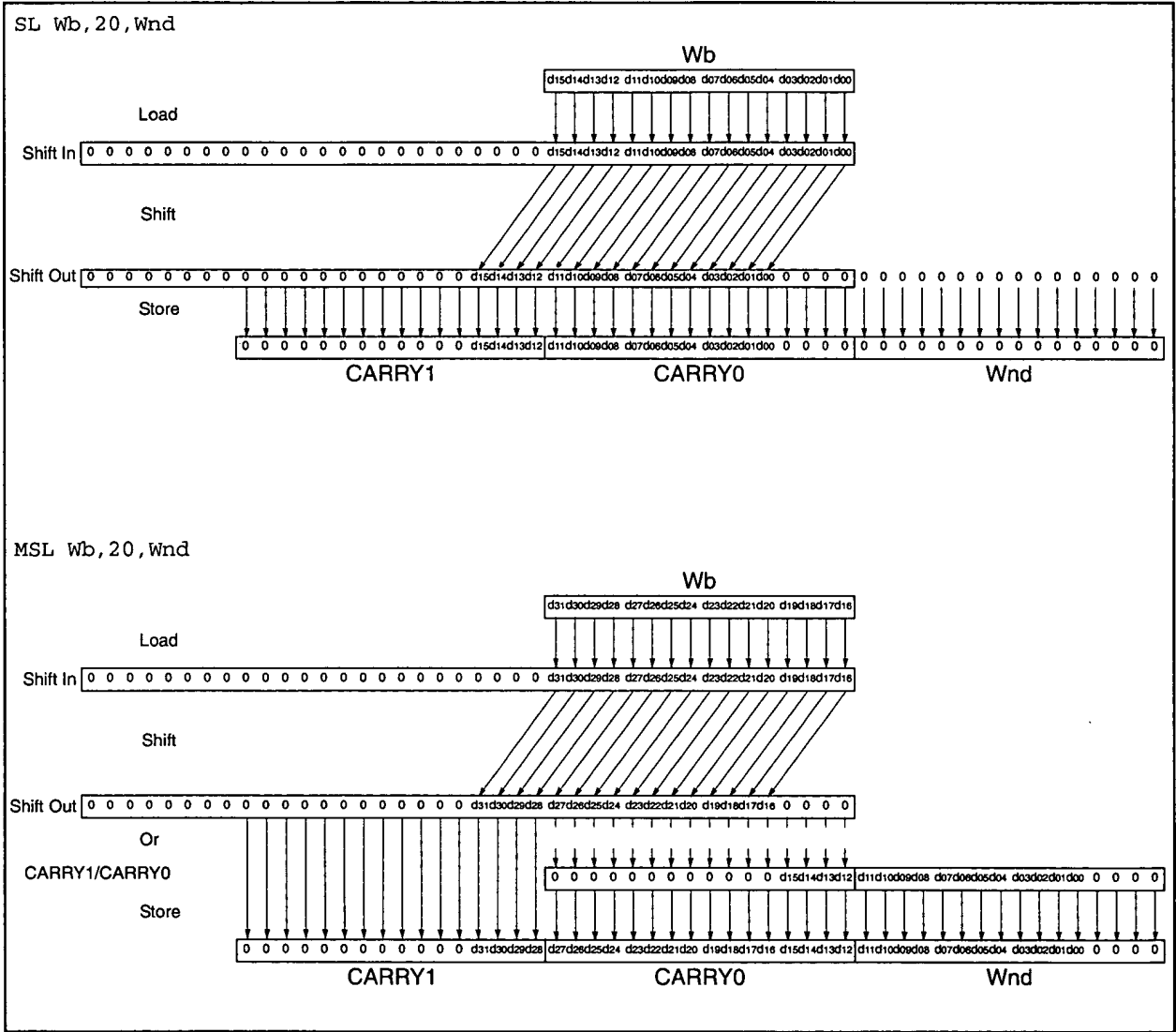


Figure 5-13 provides an example where the shift amount is 16 or more. Here, the Wnd destination register is aligned to the right of the source, CARRY0 is aligned with the source and the CARRY1 register contains the shift out results. When the next 16-bit word is shifted, the results are OR'ed with the contents of the CARRY1 and CARRY0 register, providing the shift in

from the previous shift. The SLMK instruction may be repeated for each 16-bit segment of the multi-word shift.

Note the shifter is shifting (20-16), making the shift equivalent to the previous example. When the instruction detects a shift value greater than 15, it is only necessary to realign the result registers and perform a smaller shift.

FIGURE 5-13: Multi-Word Left Shift by 20 Instruction Execution



5.15.2 32-BIT RIGHT SHIFTS

The multi-word right shift instructions are similar to the left shifts. Figure 5-14 provides an example where the shift amount is 15 or less. The Wnd destination register is aligned with the source and the CARRY1 register contains the shift out results. The CARRY0 register is

unused and remains cleared. When the next 16-bit word is shifted, the results are OR'ed with the contents of the CARRY1 register, providing the shift in from the previous shift. The SLMK instruction may be repeated for each 16-bit segment of the multi-word shift.

FIGURE 5-14: Multi-Word Right Shift by 4 Instruction Execution

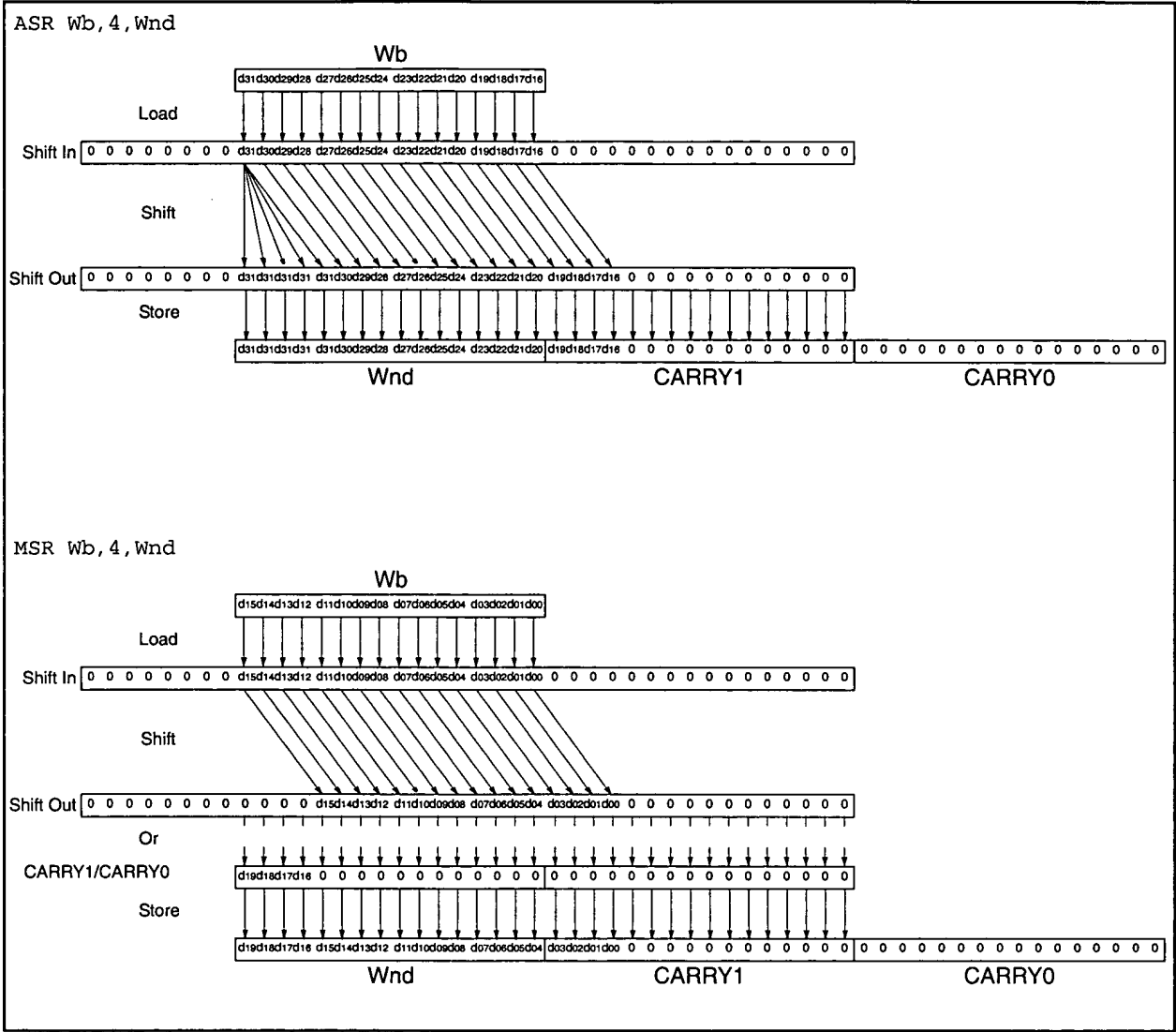
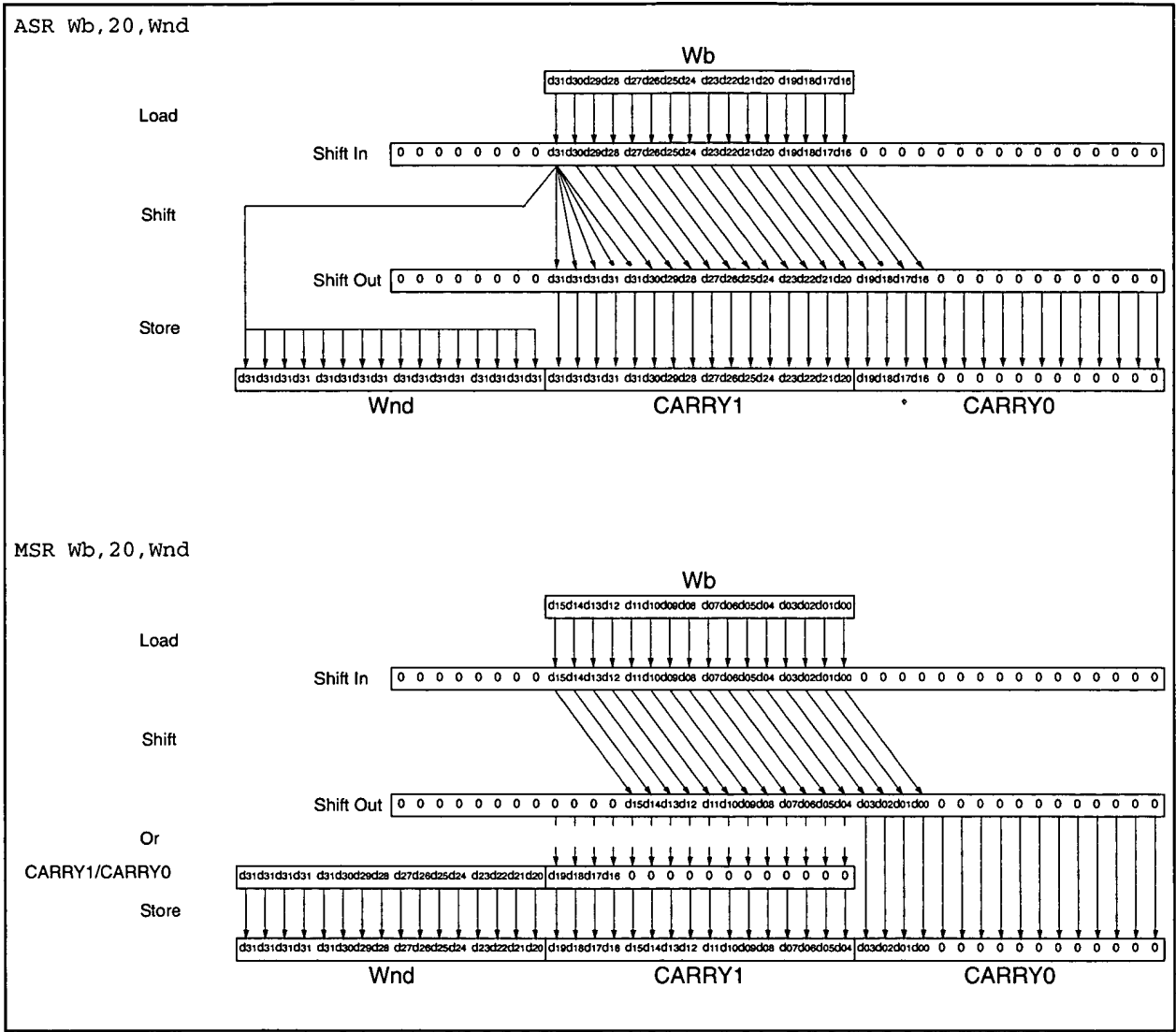


Figure 5-15 provides an example where the shift amount is 16 or more. Here, the Wnd destination register is aligned to the left of the source, CARRY1 is aligned with the source and the CARRY0 register contains the shift out results. When the next 16-bit word is shifted, the results are OR'ed with the contents of the CARRY1 and CARRY0 register, providing the shift in

from the previous shift. The SLMK instruction may be repeated for each 16-bit segment of the multi-word shift.

Note that the examples given show arithmetic shifts. If logical shifts are used, zeros would replace the sign bits.

FIGURE 5-15: Multi-Word Right Shift by 20 Instruction Execution



00000000000000000000000000000000

### 5.15.3 16-BIT SHIFTS

The ASR, LSR and SL instructions allow for shifts of 16-bit words. The shift value should be limited to 15 positions by the user for useful results.

### 5.15.4 MULTI-WORD SHIFTS ON WORDS LONGER THAN 32 BITS

The MSL and MSR instructions allow for shifts of words greater than 32 bits. This may be useful for IP addresses or encryption keys.

Note that the shift is still limited up to 31 positions.

For example, to shift a 64 bit word:

- ; W3...W0 - source word (ms...ls)
- ; W7...W4 - destination word (ms...ls)
- ; W8 - shift value (0..31)

```
Code:  LSR    W3,W8,W7
        MSR    W2,W8,W6
        MSR    W1,W8,W5
        MSR    W0,W8,W4
```

### 5.15.5 MULTI-WORD ROTATES

Because the CARRY registers are readable, the multi-word shift instructions may be used for rotates.

For example, to left rotate a 16 bit word:

```
; W1 - source word
; W0 - destination word (default Ww)
; W8 - rotate value (0..15)
Code:  SL     W1,W8,W0
        IOR    CARRY0,Ww
```

For example, to left rotate a 32 bit word:

```
; W1...W0 - source word (ms...ls)
; W3...W2 - destination word (ms...ls)
; W4 - rotate value (0..31)
; W5,W6 - temporaries
Code:  SL     W0,W4,W2
        MSL    W1,W4,W3
        MOV    CARRY0,W5
        MOV    CARRY1,W6
        IOR    W5,W2,W2 ;carry0+dest(ls)
        IOR    W6,W3,W3 ;carry1+dest(ms)
```

Using the MSL and MSR instructions, rotates of greater word lengths may be achieved.

## 5.16 DSP Data Formats

### 5.16.1 INTEGER AND FRACTIONAL DATA

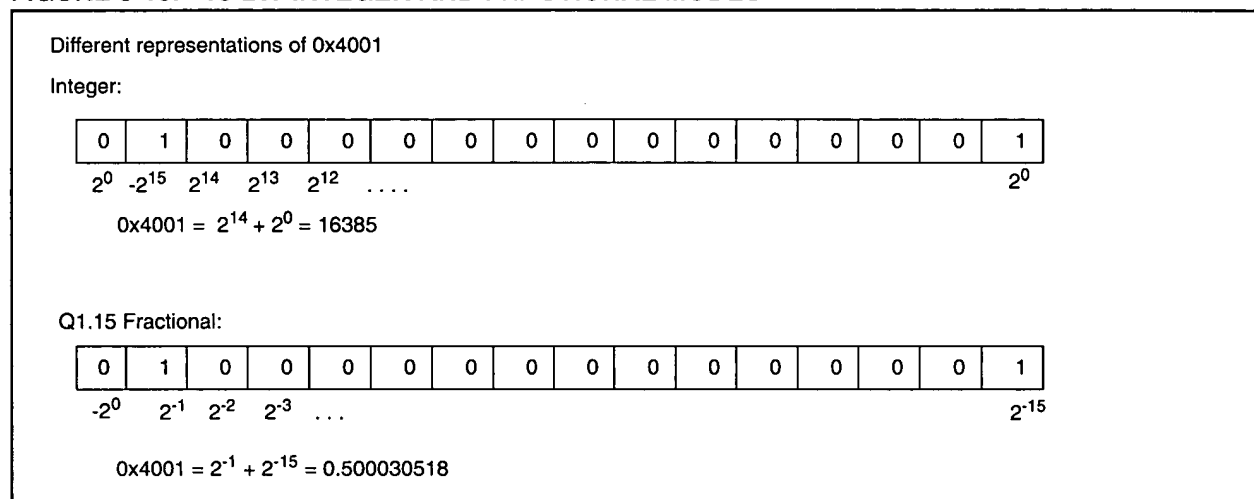
The dsPIC DSP core supports integer and fractional data operations. Data format selection is made by the IF bit in the DSP control register CORCON<0>. Setting this bit to "1" selects integer mode; setting this bit to "0" selects fractional mode.

Integer data is inherently represented as a signed two's-complement value, where the MSB is defined as a sign bit. Generally speaking, the range of an N-bit two's complement integer is  $-2^{N-1}$  to  $2^{N-1}-1$ . For a 16-bit integer, the data range is  $-32768$  (0x8000) to

32767 (0x7FFF), including 0 (see Figure 1). For a 32-bit integer, the data range is  $-2,147,483,648$  (0x8000 0000) to 2,147,483,645 (0x7FFF FFFF).

When the dsPIC is in fractional mode, data is represented as a two's complement fraction where the MSB is defined as a sign bit and the radix point is implied to lie just after the sign bit (Q1.X format). The range of an N-bit two's complement fraction with this implied radix point is  $-1.0$  to  $(1-2^{1-N})$ . For a 16-bit fraction, the Q1.15 data range is  $-1.0$  (0x8000) to 0.999969482 (0x7FFF), including 0 (see Figure 1) and has a precision of  $3.01518 \times 10^{-5}$ . In fractional mode, the 16x16 dsPIC multiplier generates a Q1.31 product which has a precision of  $4.65661 \times 10^{-10}$ .

FIGURE 5-16: 16-BIT INTEGER AND FRACTIONAL MODES



### 5.16.2 SUPER SATURATION MODE

The SATMOD bit, CORCON<3>, enables Super Saturation mode and expands the dynamic range of the accumulators by using 8 guard bits. When the SATMOD bit is set to "1", Super Saturation mode is enabled and the 40-bit accumulators support an integer range of  $-5.498 \times 10^{11}$  (0x80 0000 0000) to  $5.498 \times 10^{11}$  (0x7F FFFF FFFF). In fractional mode, the guard bits of the accumulator do not modify the location of the radix point and the 40-bit accumulators use Q9.31 fractional format. Note that all fractional operation results are stored in the 40-bit accumulator justified with a Q1.31 radix point. As in integer mode, the guard bits merely increase the dynamic range of the accumulator. Q9.31 fractions have a range of  $-256.0$  (0x80 0000 0000) to  $(256.0 - 4.65661 \times 10^{-10})$  (0x7F FFFF FFFF). See Section 2.3.3 of the Core DOS for a description of the dsPIC overflow and saturation modes.

## 5.17 Scaling and Normalizing With FBCL Instruction

To minimize quantization errors that are associated with data processing using DSP instructions, it is important to utilize the complete available resolution of the dsPIC register set. This may require scaling data up to avoid underflows (i.e., when processing data from a 12-bit ADC) or scaling data down to avoid overflows (i.e., when sending data to a 10-bit DAC). The scaling which must be performed to minimize quantization errors depends on the dynamic range of the input data which is operated on, and the requirements of the dynamic range of the output data. At times these conditions may be known apriori and fixed scaling may be employed. Other times, scaling conditions may be not be fixed or known, and then dynamic scaling must be used to process data.

The Find First Bit Change Left (FBCL) instruction can efficiently be used to perform dynamic scaling. The FBCL function determines the exponent of the byte or word which it operates on (namely the amount which the value may be shifted before overflowing), and stores the exponent such that it may be used to later scale the value by shifting. The exponent is determined by detecting the first bit change starting from the sign bit and working towards the LSB. Table 5-9 shows data with various dynamic ranges, their exponents, and the value after scaling each data to maximize the dynamic range.

**TABLE 5-9: SCALING EXAMPLES**

Data Value	Exponent	Scaled Value for Max Dynamic Range (Data Value << Exponent)
0x0001	14	0x4000
0x0002	13	0x4000
0x0004	12	0x4000
0x0100	6	0x4000
0x0101	6	0x4040
0x01FF	6	0x7FC0
0x0806	3	0x4030
0x2007	1	0x400E
0x4800	0	0x4800
0x7000	0	0x7000
0x8000	0	0x8000
0x900A	0	0x900A
0xE001	2	0x8004
0xFF07	7	0x8380
0xFFFF	0	0xFFFF*
*A "hole" where FBCL fails to detect the correct exponent		

As a practical example, assume that block processing is performed on a sequence of data with very low dynamic range stored in Q1.15 fractional format. To minimize quantization errors, the data may be scaled up to prevent any quantization loss which may occur as it is processed. The FBCL instruction can be executed on the sample with the largest magnitude to determine the optimal scaling value for processing the data. Note that scaling the data up is performed by left shifting the data (see Section 2.2 of the Core DOS for a description of the Barrel Shifter).

This is demonstrated with the code snippet below.

```
; assume W0 contains the largest absolute value of the data block
; assume W4 points to the beginning of the data block
; assume the block of data contains BLOCK_SIZE words

; determine the exponent to use for scaling
FBCL    W0, W2 ; store exponent in W2

; scale the entire data block by the optimal amount before processing
DO      SCALE_LOOP, BLOCK_SIZE
MOV     [W4], W1 ; move the next data sample to W1
SLW     W1, W2, W3 ; shift W1 by W2 bits and store to W3
SCALE_LOOP:
MOV     W3, [W4]++ ; store scaled input (overwrite original)

; now process the data
; (processing block goes here)
```

09B70457 0001001



## 5.18 Accumulator Normalization With FBCL

The process of scaling a quantized value for its maximum dynamic range is known as normalization (the data in the third column in Table 5-9 contains normalized data). Accumulator normalization is a technique used to ensure that the accumulator is properly aligned before storing data from the accumulator, and the FBCL instruction facilitates this function.

The two 40-bit accumulators each have 8 guard bits which expand the accumulator from Q1.31 to Q9.31 when operating in Super Saturation mode (see Section 1.1). Even in Super Saturation mode the Store Accumulator (SAC) instruction only stores 16-bit data (in Q1.15 format) from ACC<31:16>.

Proper data alignment for storing the contents of the accumulator may be achieved by scaling the accumulator down if the guard bits are in use, or scaling the accumulator up if all of the accumulator high bits are not being used. To perform such scaling, the FBCL instruction must operate on the guard bits in byte mode and it must operate on the high accumulator in word mode. If a shift is required, the ALU's 40-bit shifter is employed using the SFTAC instruction to perform the scaling. Listed below is a code snippet for accumulator normalization.

```
; assume an operation in ACCA has just completed (status bits are intact)
; assume the processor is in super saturation mode
; assume W4 points to the ACCA guard byte (0x44)
; assume W5 points to the ACCA high word (0x42)

    BOA      FBCL_GUARD; if overflow we right shift
FBCL_HI:
    FBCL     [W5], W0 ; extract exponent for left shift
    BRA      SHIFT_ACC; branch to the shift
FBCL_GUARD:
    FBCL.B   [W4], W0 ; extract exponent for right shift
    ADDLS.B  W0, 8, W0; adjust the sign for right shift
SHIFT_ACC:
    SFTAC    W0      ; shift the accumulator to normalize

<code assumes that negative values are returned by FBCL to facilitate scaling up>
```

5.19 DO operations

The DO instructions implement simple looping. The instruction will execute a set of instructions a certain number of times. The loop count is selected with a constant or a W register. The loop will be executed n+1 times. For a W register, only the LS 14-bits are significant. The DO instruction loads the LSR register with the value of the PC after the DO instruction. It adds the loop offset to that PC and loads that value to the LER register. It then continues to execute code starting with PC+2 until the PC matches the LER. When PC matches LER, the loop count is compared to negative. If not, the PC is loaded with the LSR value to branch back to the loop start. The loop count is decremented.

When the loop count compares negative, the next sequential instruction executes.

The instructions in the loop need not be consecutive.

FIGURE 5-17: DO OPERATION

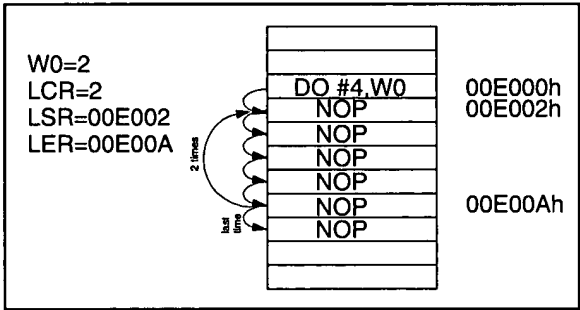
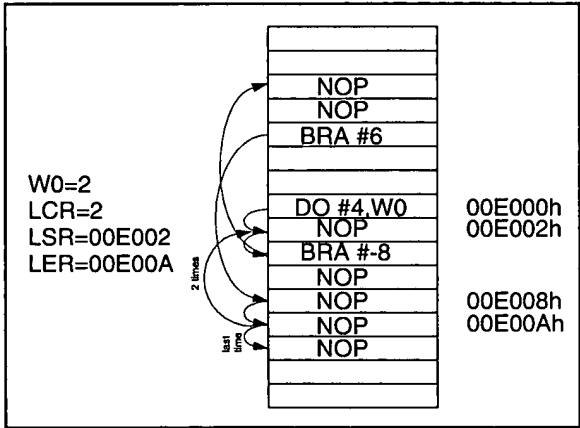


FIGURE 5-18: DO OPERATION



## 6.0 INSTRUCTION DESCRIPTIONS

The following instruction descriptions are sorted alphabetically. They are sorted and indexed by the "PLA mnemonic".

Each description lists the "PLA mnemonic" as the header. The assembly syntax lists the "assembler mnemonic" and then all of the variations of the parameters

as optional fields. The operands, a short description of the operation and the status affected follow. The bit encoding is listed. A detailed verbal description describes the operation of the instruction. Examples are shown for each of the major operand variants.

Table 6-1 lists the symbols used in the instruction descriptions.

**TABLE 6-1: SYMBOLS USED IN ROADRUNNER OPCODE DESCRIPTIONS**

Field	Description
{ }	Optional field or operation
[text]	Means "the location addressed by text"
(text)	Means "content of text"
#text	Means literal defined by "text"
text1 ∈ {text2, text3, ...}	text1 must be in the set of text2, text3, ...
none	field does not require an entry, may be blank
{label:}	Optional Label name
label	Translates to a literal representing the location of the label name
<n:m>	Register bit field
lit1	1-bit unsigned literal ∈ {0,1}
lit4	4-bit unsigned literal ∈ {0...15}
lit5	5-bit unsigned literal ∈ {0...31}
slit5	5-bit signed literal ∈ {-16...15}
slit10	10-bit signed literal ∈ {-512...511}
lit14	14-bit unsigned literal ∈ {0...16384}
lit16	16-bit unsigned literal ∈ {0...65535}
slit16	16-bit signed literal ∈ {-32768...32767}
lit23	23-bit unsigned literal ∈ {0...8388608}; LSB must be 0
bit3	3-bit bit selection field (used in byte addressed instructions) ∈ {0...7}
bit4	4-bit bit selection field (used in word addressed instructions) ∈ {0...15}
.w	Word mode selection (default)
.b	Byte mode selection
.s	Shadow register select
f	File register address ∈ {0000h...1FFFh}
d	File register destination d ∈ {ww, none}
Ww	Default W working register (used in file register instructions)
Wn	One of 16 working registers ∈ {W0..W15}
Wns	One of 16 source working registers ∈ {W0..W15}
Wnd	One of 16 destination working registers ∈ {W0..W15}
Wb	Base W register ∈ {W0..W15}
Ws	Source W register ∈ { Ws, [Ws], [Ws]++, [Ws]--, [Ws++] }
Wd	Destination W register ∈ { Wd, [Wd], [Wd]++, [Wd]--, [Wd++] }
Wso	Source W register ∈ { Wns, [Wns], [Wns]++, [Wns]--, [Wns--], [Wns+Wb], [Wns+slit5] }
Wdo	Destination W register ∈ { Wnd, [Wnd], [Wnd]++, [Wnd]--, [Wnd--], [Wnd+Wb], [Wnd+slit5] }

Field	Description
$W_m * W_m$	Multiplicand and Multiplier W register for Square instructions $\in \{W0*W0, W1*W1, W2*W2, W3*W3\}$
$W_m * W_n$	Multiplicand and Multiplier W register for DSP instructions $\in \{W0*W1, W0*W2, W0*W3, W1*W2, W1*W3, W2*W3\}$
$W_x$	X data space prefetch address register for DSP instructions $\in \{[W4] += 6, [W4] += 4, [W4] += 2, [W4], [W4] -= 6, [W4] -= 4, [W4] -= 2, [W5] += 6, [W5] += 4, [W5] += 2, [W5], [W5] -= 6, [W5] -= 4, [W4] -= 2, [W5 + W8], \text{none}\}$
$W_y$	Y data space prefetch address register for DSP instructions $\in \{[W6] += 8, [W6] += 4, [W6] += 2, [W6], [W6] -= 6, [W6] -= 4, [W6] -= 2, , [W7] += 8, [W7] += 4, [W7] += 2, [W7], [W7] -= 6, [W7] -= 4, [W7] -= 2, , [W7 + W8], \text{none}\}$
$W_{xp}$	X data space prefetch destination register for DSP instructions $\in \{W0..W3\}$
$W_{yp}$	Y data space prefetch destination register for DSP instructions $\in \{W0..W3\}$
AWB	Accumulator write back destination address register $\in \{W9, [W9]++\}$
PC	Program Counter
PCL	Program Counter Low Byte
PCH	Program Counter High Byte
PCU	Program Counter Upper Byte
PCLATH	Program Counter High Byte Latch
PCLATU	Program Counter Upper Byte Latch
OA, OB, SA, SB	DSC status bits: ACCA Overflow, ACCB Overflow, ACCA Saturate, ACCB Saturate
C, DC, N, OV, SZ, Z	ALU status bits: Carry, Digit Carry, Negative, Overflow, Sticky-Zero, Zero

[illegible]

Syntax:	{label:}	ADD{.b}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws]++,	[Wd]++
				[Ws]--,	[Wd]--
				[Ws++] ,	[Wd++]
				[Ws--] ,	[Wd--]

**Status Affected:** C, DC, N, OV, Z

Encoding:	0100	0www	wBqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'w' bits select the address of the base register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select source address mode 2.
- The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Cycles: 1

## Examples

**Example1**                  ADD      W5,W6,W7        ; Add

### Before Instruction

### After Instruction

### Add ACCA to ACCB

Syntax:	{label:}	ADD	A
			B

Operands: none

Operation:  $(ACCA) + ACCB \rightarrow ACC(A \text{ or } B)$

Status Affected: OA, OB, SA, SB

**Encoding:**

1100	1011	A000	0000	0000	0000
------	------	------	------	------	------

Description:	Add ACCA to ACCB and write results to selected accumulator.
--------------	---

The 'A' bits specify the destination accumulator.

**Words:** 1

Cycles: 1

## Examples

Example1	ADD	B	; Add ACCA to ACCB, result to ACCB
----------	-----	---	------------------------------------

Before Instruction

### After Instruction

0982045-060101

03070457-060101

ADDAC

16-Bit Signed Add to Accumulator

Syntax: {label:} ADD A, Wns, [, Slit4]  
B, [Wns],  
[Wns]++  
[Wns]--  
[Wns--],  
[Wns+Wb],  
[Wns+lit5]

Operands: Wns ∈ [W0 ... W15];  
Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]  
Slit4 ∈ [-8 ... +7]

Operation: (ACC) + Shift<sub>Slit4</sub>(Extend(Wns)) → ACC

Status Affected: OA, OB, SA, SB

Encoding:	1100	1001	Awww	wrrr	rggg	ssss
-----------	------	------	------	------	------	------

Description: The term contained at the effective address is assumed to be Q15 fractional data and is automatically sign-extended and zero-backfilled prior to the operation.

Optionally shift the term, then add the term to accumulator.

The 'A' bits specify the destination accumulator.  
The 's' bits specify the source register Wns.  
The 'g' bits select source address mode 3.  
The 'w' bits specify the offset amount lit5 OR the offset register Wb.  
The 'r' bits encode the optional operand Slit4 which determines the amount of the accumulator preshift; if the operand Slit4 is absent, a 0 is encoded.

See Table 1-7 for modifier addressing information.

**Note:** Positive values of operand Slit4 represent arithmetic shift right.  
Negative values of operand Slit4 represent shift left.

Words: 1  
Cycles: 1

Examples

Example1 ADD A,W5,# 3 ; Shift W5 right 3 bits, add to accumu-  
lator A  
  
Before Instruction  
  
After Instruction



### Add Wb and Ws with Carry

Syntax:	{label:}	ADDC{.b}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws]++,	[Wd]++
				[Ws]--,	[Wd]--
				[Ws++] ,	[Wd++]
				[Ws--],	[Wd--]

Operands:  $Wb \in [W0 \dots W15]$ ;  $Ws \in [W0 \dots W15]$ ;  $Wd \in [W0 \dots W15]$

Operation:  $(Wb) + (Ws) + (C) \rightarrow Wd$

**Status Affected:** C, DC, N, OV, Z

**Encoding:**

0100	1www	wBqq	qddd	dppp	ssss
------	------	------	------	------	------

**Description:**

Add the contents of the source register Ws and the contents of the base register Wb and the Carry bit and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.

The 's' bits select the address of the source register.

The 'w' bits select the address of the base register.

The 'd' bits select the address of the destination register.

The 'p' bits select source address mode 2.

The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

**Example1**                  ADDC    W5,W6,W7       ; Add

### Before Instruction

### After Instruction

ADDCLS

Add Wb and Short Literal with Carry

Syntax:

{label:}    ADDC{.b}    Wb,    lit5,    Wd

[Wd]

[Wd]++

[Wd]--

[Wd++]

[Wd--]

Operands:

Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]

Operation:

(Wb) + lit5 + (C) → Wd

Status Affected:

C, DC, N, OV, Z

Encoding:

0100	1www	wBqq	qddd	d11k	kkkk
------	------	------	------	------	------

Description:

Add the contents of the base register Wb, the literal operand and the Carry bit; and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.  
The 'w' bits select the address of the base register.  
The 'k' bits provide the literal operand, a five-bit integer number.  
The 'd' bits select the address of the destination register.  
The 'q' bits select destination address mode 2.

See Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:

1

Cycles:

1

Examples

Example1

ADDC    W5,#12,W7    ; Add

Before Instruction

After Instruction

### Add Literal to Wn with Carry

Syntax: {label:} ADDC{.b} Slit10, Wn

**Operands:** Slit10  $\in [-512 \dots 511]$ ; Wn  $\in [W0 \dots W15]$

Operation:  $\text{Slit10} + (\text{Wn}) + (\text{C}) \rightarrow \text{Wn}$

Status Affected: C, DC, N, OV, Z

Encoding:

1011	0000	1Bkk	kkkk	kkkk	dddd
------	------	------	------	------	------

**Description:**

Add the literal operand to the contents of the working register Wn and the Carry bit and place the result in the working register Wn.

The 'B' bit selects byte or word operation.

The 'd' bits select the address of the working register.

The 'k' bits specify the literal operand, a signed 10-bit number.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

**Example1**                      **ADDC    #123,W7                      ; Add w/ carry**

### Before Instruction

### After Instruction

[illegible]

# ADDLS

### Add Wb and Short Literal

Syntax:	{label:}	ADD{.b}	Wb,	lit5,	Wd
					[Wd]
					[Wd]++
					[Wd]--
					[Wd++]
					[Wd--]

**Operands:** Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]

Operation:  $(Wb) + lit5 \rightarrow Wd$

Status Affected: C, DC, N, OV, Z

Encoding:	0100	0www	wBqq	qddd	d11k	kkkk
-----------	------	------	------	------	------	------

**Description:** Add the contents of the source register Ws and the literal operand and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.

The 'w' bits select the address of the base register.

The 'k' bits provide the literal operand, a five-bit integer number.

The 'd' bits select the address of the destination register.

The 'q' bits select destination address mode 2.

See Table 1-6 for modifier addressing information.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

**Words:** 1

Cycles: 1

## Examples

**Example1**                      ADD      W5,#12,W7              ; Add

### Before Instruction

### After Instruction

### Add Literal to Wn

Syntax: {label:} ADD{.b} Slit10, Wn

**Operands:** Slit10  $\in [-512 \dots 511]$ ; Wn  $\in [W0 \dots W15]$

Operation:  $\text{Slit10} + (W_n) \rightarrow W_n$

Status Affected: C, DC, N, OV, Z

**Encoding:**

1011	0000	0Bkk	kkkk	kkkk	dddd
------	------	------	------	------	------

**Description:**

Add the literal operand to the contents of the working register Wn and place the result in the working register Wn.

The 'B' bit selects byte or word operation.

The 'd' bits select the address of the working register.

The 'k' bits specify the literal operand, a signed 10-bit number.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

**Example1**                      **ADD    #123,W7                      ; Add**

### Before Instruction

### After Instruction

[illegible]

### Add f and Ww

Syntax:                    {label:}        ADD{.b}                f                {,Ww}

Operands:  $f \in [0 \dots 8191]$

Operation: (f) + (Ww) → destination designated by D

Status Affected: C, DC, N, OV, Z

**Encoding:**

1011	0100	0BDf	ffff	ffff	ffff
------	------	------	------	------	------

**Description:**

Add the contents of the working register and the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.

The 'D' bit selects the destination.

The 'f' bits select the address of the file register.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

Example1                      ADD      RAM135, Ww      ; Add

**Before Instruction**

### After Instruction

**DEVELOPMENT OF A**

# ADDWFC

Add f and Carry bit and Ww

Syntax: {label:} ADDC{.b} f {,Ww}

Operands:

f ∈ [0 ... 8191]

Operation:

(f) + (Ww) + (C)→ destination designated by D

Status Affected:

C, DC, N, OV, Z

Encoding:

1011	0100	1BDE	ffff	ffff	ffff
------	------	------	------	------	------

Description:

Add the contents of the working register and the carry flag and the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination.  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:

1

Cycles:

1

## Examples

Example1

ADDC

RAM135, Ww

; Add

Before Instruction

After Instruction

## And Wb and Ws

Syntax:	{label:}	AND{.b}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws]++,	[Wd]++
				[Ws]--,	[Wd]--
				[Ws+++],	[Wd+++]
				[Ws---],	[Wd---]

Operands:  $Wb \in [W0 \dots W15]$ ;  $Ws \in [W0 \dots W15]$ ;  $Wd \in [W0 \dots W15]$

Operation:  $(Wb).AND.(Ws) \rightarrow Wd$

Status Affected: N, Z

Encoding:

0110	0www	wBqq	qddd	dppp	ssss
------	------	------	------	------	------

**Description:**

Compute the AND of the contents of the source register Ws and the contents of the base register Wb and place the result in the destination register Wd.

- The 'B' bit selects byte or word operation.

The 's' bits select the address of the source register.

The 'w' bits select the address of the base register.

The 'd' bits select the address of the destination register.

The 'p' bits select source address mode 2.

The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

Example1                      AND            W5,W6,W7            ; And

### Before Instruction

### After Instruction

0987042



# ANDLS

## AND Wb and Short Literal

Syntax:	{label:}	AND{.b}	Wb,	lit5,	Wd
					[Wd]
					[Wd]++
					[Wd]--
					[Wd++]
					[Wd--]

Operands:	Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]				
Operation:	(Wb).AND.lit5 → Wd				
Status Affected:	N, Z				
Encoding:	0110	0www	wBqq	qddd	
Description:	Compute the AND of the contents of the base register				

The 'B' bit selects byte or word operation.  
The 'w' bits select the address of the base register.  
The 'k' bits provide the literal operand, a five-bit integer number.  
The 'd' bits select the address of the destination register.  
The 'q' bits select destination address mode 2.

See Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:	1
Cycles:	1

### Examples

Example1	AND W5,#12,W7 ; AND
	Before Instruction
	After Instruction

FOR90"Z540Z860

# ANDLW

AND Literal and Wd

Syntax: {label:} AND{.b} Slit10, Wn

Operands: Slit10 ∈ [-512 ... 511]; Wn ∈ [W0 ... W15]

Operation: Slit10.AND.(Wn) → Wn

Status Affected: N, Z

Encoding: 

1011	0010	0Bkk	kkkk	kkkk	dddd
------	------	------	------	------	------

Description: Compute the AND of the literal operand and the contents of the working register Wn and place the result in the working register Wn.

The 'B' bit selects byte or word operation.  
The 'd' bits select the address of the working register.  
The 'k' bits specify the literal operand, a signed 10-bit number.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1                      AND    #123,W7                      ; AND

Before Instruction

After Instruction

**060107**

Syntax: {label:} AND{.b} f {,Ww}

Operation: (f).AND.(Ww) → destination designated by D

Encoding:	1011	0110	0BDF	ffff	ffff	ffff
-----------	------	------	------	------	------	------

The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination.  
The 'f' bits select the address of the file register.

Words: 1  
Cycles: 1

### After Instruction

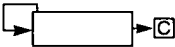
# ASR

## Arithmetic Shift Right Ws

Syntax:	{label:}	ASR{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++] ,	[Wd++]
			[Ws--],	[Wd--]

Operands: Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]

Operation: For word operation:  
(Ws<15>) → Wd<15>, (Ws<15>) → Wd<14>,  
(Ws<14:1>) → Wd<13:0>, (Ws<0>) → C  
For byte operation:  
(Ws<7>) → Wd<7>, (Ws<7>) → Wd<6>,  
(Ws<6:1>) → Wd<5:0>, (Ws<0>) → C



Status Affected: C, N, OV, Z

Encoding:	1101	0001	1Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Shift the contents of the source register Ws one bit to the right and place the result in the destination register Wd. Shift the MSB back into itself. The Carry Flag is set if the LSB of Ws is '1'.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select source address mode 2.
- The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

### Examples

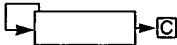
Example1 ASR W5,W6 ; Arithmetic shift right  
Before Instruction  
  
After Instruction

# ASRF

## Arithmetic Shift Right f

Syntax: {label:} ASR{.b} f {,Ww}

Operands: f ∈ [0 ... 8191]  
 Operation: For word operation:  
 (f<15>) → Dest<15>, (f<15>) → Dest<14>  
 (f<14:1>) → Dest<13:0>, (f<0>) → C  
 For byte operation:  
 (f<7>) → Dest<7>, (f<7>) → Dest<6>,  
 (f<6:1>) → Dest<5:0>, (f<0>) → C



Status Affected: C, N, OV, Z

Encoding:	1101	0101	1BDf	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description: Shift the contents of the file register f one bit to the right through the carry flag and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.  
 The 'D' bit selects the destination.  
 The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
 Cycles: 1

### Examples

Example1 ASR RAM135, Ww ; Arithmetic shift right  
 Before Instruction  
  
 After Instruction

# ASRK

## Arithmetic Shift Right by Short Literal

Syntax: {label:} ASR Wb, lit5, Wnd

Operands: Wb ∈ [W0 ... W15]; lit5 ∈ [0...31]; Wnd ∈ [W0 ... W15]

Operation: lit5<3:0>→Shift\_Val

0→Shift\_In<39:32>

Wb<15:0>→Shift\_In<31:16>

0→Shift\_In<15:0>

0→Shift\_Out<39:32>

Shift\_In<31>→Shift\_Out<32:32-Shift\_Val>

Shift\_In<31:Shift\_Val>→Shift\_Out<31-Shift\_Val:0>

If lit5<4>==0: (less than 16)

Shift\_Out<31:16>→Wnd

Shift\_Out<15:0>→CARRY1

0→CARRY0

If lit5<4>==1: (16 or greater)

Shift\_Out<31:31>→Wnd<15:0>

Shift\_Out<31:16>→CARRY1

Shift\_Out<15:0>→CARRY0

Status Affected: C,SZ,Z

Encoding:

1101	1110	1www	wddd	d11k	kkkk
------	------	------	------	------	------

Description:

Arithmetic shift right the contents of the source register Wb by lit5 bits (up to 31 positions), placing the result in the destination register Wnd. Bits that are shifted beyond the rightmost position of the source are stored in the CARRY1 and CARRY0 registers.

The Z and SZ bits will be set if the value placed in Wnd is zero and cleared otherwise. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.

**Note:** This instruction operates in word mode only.

Words: 1

Cycles: 1

### EXAMPLES:

ASRW

Arithmetic Shift Right by Wns

Syntax: (label:) ASR Wb, Wns, Wnd

Operands: Wb ∈ [W0 ... W15]; Wns ∈ [W0 ...W15]; Wnd ∈ [W0 ... W15]

Operation: Wns<3:0>→Shift\_Val

0→Shift\_In<39:32>  
Wb<15:0>→Shift\_In<31:16>  
0→Shift\_In<15:0>

0→Shift\_Out<39:32>  
Shift\_In<31>→Shift\_Out<32:32-Shift\_Val>  
Shift\_In<31:Shift\_Val>→Shift\_Out<31-Shift\_Val:0>

If Wns<4>==0: (less than 16)  
Shift\_Out<31:16>→Wnd  
Shift\_Out<15:0>→CARRY1  
0→CARRY0  
If Wns<4>==1: (16 or greater)  
Shift\_Out<31:31>→Wnd<15:0>  
Shift\_Out<31:16>→CARRY1  
Shift\_Out<15:0>→CARRY0

Status Affected: C,SZ,Z

Encoding:	1101	1110	1www	wddd	d000	ssss
-----------	------	------	------	------	------	------

Description: Arithmetic shift right the contents of the source register Wb by Wns bits (up to 31 positions), placing the result in the destination register Wnd. Bits that are shifted beyond the rightmost position of the source are stored in the CARRY1 and CARRY0 registers.

The Z and SZ bits will be set if the value placed in Wnd is zero and cleared otherwise. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.

Note: This instruction operates in word mode only.

Words: 1  
Cycles: 1

EXAMPLES:

# BC

## Branch if Carry

Syntax:	{label:}	BRA	C,	Slit16
	{label:}	BRA	GEU,	

Operands:	Slit16 ∈ [-32768 ... +32767]					
Operation:	Condition = C If (condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.					
Status Affected:	None					
Encoding:	0011	0001	nnnn	nnnn	nnnn	nnnn
Description:	If the Carry bit is '1', then the program will branch.					

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2\*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words:	1
Cycles:	1 (2)

### Examples

Example1	BRA	C, label	; Branch if Carry
	Before Instruction		
	After Instruction		



# BCLR

## Bit Clear in Ws

Syntax:	{label:}	BCLR	Ws,	bit4
			[Ws],	
			[Ws]++,	
			[Ws]--,	
			[Ws++] ,	
			[Ws--],	

Operands:	bit4 ∈ [0 ... 15]; Ws ∈ [W0 ... W15]					
Operation:	0 → Ws<bit4>					
Status Affected:	None					
Encoding:	1010	0001	bbbb	0000	0ppp	ssss
Description:	Bit 'bit4' in register Ws is cleared.					

The 'b' bits select value bit4 of the bit position to be cleared.  
The 's' bits select the address of the source/destination register.  
The 'p' bits select source address mode 2.

See Table 1-6 for modifier addressing information.

**Note:** This instruction operates in word mode only.

Words:	1
Cycles:	1

### Examples

Example1	BCLR	W6, #5	; Clear bit 5 in W6
	Before Instruction		
	After Instruction		

# BCLRF

Bit Clear f

Syntax: {label;} BCLR.b f, bit3

Operands: bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]

Operation: 0 → f<bit3>

Status Affected: None

Encoding:	1010	1001	bbbf	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description: Bit 'bit3' in file register f is cleared.

The 'b' bits select value bit3 of the bit position to be cleared.  
The 'f' bits select the address of the file register.

- Note:** This instruction operates in byte mode only.
- Note:** The .b extension must be included with the opcode.

Words: 1

Cycles: 1

## Examples

Example1                    BCLR.b   RAM135, #5           ; Clear bit 5 in RAM135

Before Instruction

After Instruction

BGE

Branch if Signed Greater Than or Equal

Syntax:

{label:}

BRA

GE,

Slit16

Operands:

Slit16 ∈ [-32768 ... +32767]

Operation:

Condition = (N&&OV)!!(N&&!OV)

If (Condition), then (PC+2) + 2\*Slit16 → PC, and NOP → Instruction Register.

Status Affected:

None

Encoding:

0011	1101	nnnn	nnnn	nnnn	nnnn
------	------	------	------	------	------

Description:

If the branch condition is met, then the program will branch.

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2\*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words:

1

Cycles:

1 (2)

Examples

Example1

BRA

GE, label

; Branch if Greater Than or Equal

Before Instruction

After Instruction

# BGT

## Branch if Signed Greater Than

Syntax: {label:} BRA GT, Slit16

**Operands:** Slit16 ∈ [-32768 ... +32767]

Operation: Condition = (I<sub>Z</sub> & N & O<sub>V</sub>) || (I<sub>Z</sub> & !N & !O<sub>V</sub>);  
If (Condition), then (PC+2) + 2\*Slit16 → PC, and NOP → Instruction Register.

Status Affected:           None

Encoding:	0011	1100	nnnn	nnnn	nnnn	nnnn
-----------	------	------	------	------	------	------

<b>Description:</b>	If the branch condition is met, then the program will branch.
---------------------	---

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be  $(PC+2) + 2*Slit16$ . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

**Words:** 1

Cycles: 1 (2)

## Examples

Example1	BRA	GT, label	; Branch if Greater Than
Before Instruction			

### After Instruction

02820101

BGTU

Branch if Unsigned Greater Than

Syntax: {label:} BRA GTU, Slit16

Operands: Slit16 ∈ [-32768 ... +32767]  
Operation: Condition = (C&&!Z);  
If (Condition), then (PC+2) + 2\*Slit16 → PC, and NOP → Instruction Register.  
Status Affected: None  
Encoding: 

0011	1110	nnnn	nnnn	nnnn	nnnn
------	------	------	------	------	------

  
Description: If the branch condition is met, then the program will branch.

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2\*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.  
  
The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1  
Cycles: 1 (2)

Examples

Example1                      BRA      GTU, label                      ; Branch if Unsigned Greater Than  
Before Instruction  
  
After Instruction

# BLE

### Branch if Signed Less Than or Equal

**Syntax:** {label:} BRA LE, Slit16

Operands: Slit16  $\in [-32768 \dots +32767]$

**Operation:** Condition = Z||N&&!OV||(!N&&OV);  
If (Condition), then (PC+2) + 2\*Slit16 → PC, and NOP → Instruction Register.

Status Affected:           None

Encoding:	0011	0100	nnnn	nnnn	nnnn	nnnn
-----------	------	------	------	------	------	------

**Description:** If the branch condition is met, then the program will branch.

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be  $(PC+2) + 2*Slit16$ . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

**Words:** 1

Cycles: 1 (2)

## Examples

Example1	BRA	LE, label	; Branch if Less Than or Equal
Before Instruction			

### After Instruction

09870457 060101  
T0T090 25402860

BLEU

Branch if Unsigned Less Than or Equal

Syntax: {label:} BRA LEU, Slit16

Operands: Slit16 ∈ [-32768 ... +32767]  
Operation: Condition = !CltZ;  
If (Condition), then (PC+2) + 2\*Slit16 → PC, and NOP → Instruction Register.  
Status Affected: None  
Encoding: 

0011	0110	nnnn	nnnn	nnnn	nnnn
------	------	------	------	------	------

  
Description: If the branch condition is met, then the program will branch.

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2\*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.  
  
The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1  
Cycles: 1 (2)

Examples

Example1                      BRA      LEU, label                      ; Branch if Unsigned Less Than or Equal  
  
Before Instruction  
  
After Instruction

# BLT

## Branch if Signed Less Than

**Syntax:** {label:} BRA LT, Slit16

**Operands:** Slit16  $\in [-32768 \dots +32767]$

Operation: Condition = (N&&!OV) || (!N&&OV);  
If (Condition), then (PC+2) + 2\*Slit16 → PC, and NOP → Instruction Register.

Status Affected: None

Encoding:	0011	0101	nnnn	nnnn	nnnn	nnnn
-----------	------	------	------	------	------	------

<b>Description:</b>	If the branch condition is met, then the program will branch.
---------------------	---

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be  $(PC+2) + 2*Slit16$ . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2)

## Examples

Example1	BRA	LT, label	; Branch if Less Than
Before Instruction			

### After Instruction



### Branch if Negative

**Syntax:** {label:} BRA N<sub>i</sub> Slit16

Operands: Slit16  $\in [-32768 \dots +32767]$

Operation: Condition = N

If (condition), then  $(PC+2) + 2*Slit16 \rightarrow PC$ , and  $NOP \rightarrow$  Instruction Register.

Status Affected: None

**Encoding:**

0011	0011	nnnn	nnnn	nnnn	nnnn
------	------	------	------	------	------

**Description:**

If the Negative Flag is '1', then the program will branch.

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be  $(PC+2) + 2*Slit16$ . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

**Words:** 1

Cycles: 1 (2)

## Examples

Example1	BRA	N, label	; Branch if Negative
----------	-----	----------	----------------------

### Before Instruction

### After Instruction

[illegible]

# BNC

## Branch if Not Carry

Syntax:	{label:}	BRA	NC,	Slit16
	{label:}	BRA	LTU,	

Operands:	Slit16 ∈ [-32768 ... +32767]					
Operation:	Condition = !C If (condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.					
Status Affected:	None					
Encoding:	0011	1001	nnnn	nnnn	nnnn	nnnn
Description:	If the Carry bit is '0', then the program will branch.					

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2\*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words:	1
Cycles:	1 (2)

### Examples

Example1	BRA	NC, label	; Branch if Not Carry
	Before Instruction		
	After Instruction		

# BNN

Branch if Not Negative

Syntax: {label:} BRA NN, Slit16

Operands: Slit16  $\in$  [-32768 ... +32767]

Operation: Condition = !N  
If (condition), then  $(PC+2) + 2*Slit16 \rightarrow PC$ , and NOP  $\rightarrow$  Instruction Register.

Status Affected: None

Encoding: 

0011	1011	nnnn	nnnn	nnnn	nnnn
------	------	------	------	------	------

Description: If the Negative Flag is '0', then the program will branch.

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be  $(PC+2) + 2*Slit16$ . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2)

## Examples

Example1 BRA NN, label ; Branch if Not Negative  
Before Instruction

After Instruction



[illegible]

## 1

**Syntax:**

```
{label:}
```

**BRA**

NZ,

Slit16

Operands:

Slit16 ∈ [-32768 ... +32767]

**Operation:**

Condition = !Z

If (condition), then  $(PC+2) + 2*Slit16 \rightarrow PC$ , and  $NOP \rightarrow \text{Instruction Register}$ .

**Status Affected:**

None

**Encoding:**

0011	1010	nnnn	nnnn	nnnn	nnnn
------	------	------	------	------	------

**Description:**

If the Zero Flag is '0', then the program will branch.

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be  $(PC+2) + 2*Slit16$ . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

**Words:**

1

**Cycles:**

1 (2)

## Examples

### Example1

BRA      NZ, label

; Branch if Not Zero

### Before Instruction

### After Instruction

---

Syntax:                    {label:}      BRA                    OA,                    Slit16

Operation: Condition = OA  
If (condition), then  $(PC+2) + 2*Slit16 \rightarrow PC$ , and NOP  $\rightarrow$  Instruction Register.

0000	1100	nnnn	nnnn	nnnn	nnnn
------	------	------	------	------	------

**Description:** If the OA Flag is '1', then the program will branch.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Cycles: 1 (2)

Example1	BRA	OA, label	; Branch if Accumulator A Overflow
Before Instruction			
After Instruction			

BOB

Branch if Overflow Accumulator B

Syntax:

{label:}

BRA

OB,

Slit16

Operands:

Slit16 ∈ [-32768 ... +32767]

Operation:

Condition = OB

If (condition), then (PC+2) + 2\*Slit16 → PC, and NOP → Instruction Register.

Status Affected:

None

Encoding:

0000	1101	nnnn	nnnn	nnnn	nnnn
------	------	------	------	------	------

Description:

If the OB Flag is '1', then the program will branch.

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2\*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words:

1

Cycles:

1 (2)

Examples

Example1

BRA

OB, label

; Branch if Accumulator B Overflow

Before Instruction

After Instruction

# BOV

## Branch if Overflow

Syntax: {label:} BRA OV, Slit16

Operands: Slit16 ∈ [-32768 ... +32767]  
 Operation: Condition = OV  
 If (condition), then (PC+2) + 2\*Slit16 → PC, and NOP → Instruction Register.  
 Status Affected: None  
 Encoding: 

0011	0000	nnnn	nnnn	nnnn	nnnn
------	------	------	------	------	------

  
 Description: If the Overflow Flag is '1', then the program will branch.

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2\*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1  
 Cycles: 1 (2)

### Examples

Example1                      BRA      OV, label                      ; Branch if Overflow

Before Instruction

After Instruction



[illegible]

## 1

Syntax: {label:} BRA Slit16

Operands: Slit16  $\in [-32768 \dots +32767]$

Operation:  $(PC+2) + 2*Slit16 \rightarrow PC$ , and NOP  $\rightarrow$  Instruction Register.

Status Affected: None

0011	0111	nnnn	nnnn	nnnn	nnnn
------	------	------	------	------	------

The program will branch unconditionally.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 2

## Examples

Example1	BRA	label	; Branch unconditionally
----------	-----	-------	--------------------------

Before Instruction

After Instruction

[illegible]

Syntax:	{label:}	BRA	Wn
---------	----------	-----	----

Encoding:	0000	0001	0110	0000	0000	ssss
-----------	------	------	------	------	------	------

The 's' bits select the address of the source register.

## Examples

### After Instruction

BSA

Branch if ACCA Saturation

Syntax: {label:} BRA SA, Slit16

Operands: Slit16 ∈ [-32768 ... +32767]  
Operation: Condition = SA  
If (condition), then (PC+2) + 2\*Slit16 → PC, and NOP → Instruction Register.  
Status Affected: None  
Encoding: 

0000	1110	nnnn	nnnn	nnnn	nnnn
------	------	------	------	------	------

  
Description: If the ACCA Saturation Flag is '1', then the program will branch.

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2\*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.  
  
The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1  
Cycles: 1 (2)

Examples

Example1                    BRA    SA, label                    ; Branch if ACCA Saturation  
Before Instruction  
  
After Instruction

### Branch if ACCB Saturation

**Syntax:** {label:} BRA SB, Slit16

**Operands:** Slit16  $\in [-32768 \dots +32767]$

Operation: Condition = SB  
if (condition), then  $(PC+2) + 2*Slit16 \rightarrow PC$ , and NOP  $\rightarrow$  Instruction Register.

Status Affected:           None

Encoding:	0000	1111	nnnn	nnnn	nnnn	nnnn
-----------	------	------	------	------	------	------

**Description:** If the ACCB Saturation Flag is '1', then the program will branch.

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be  $(PC+2) + n$ . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2)

## Examples

Example1	BRA	SB, label	; Branch if ACCB Saturation
----------	-----	-----------	-----------------------------

### Before Instruction

### After Instruction

# BSET

## Bit Set in Ws

Syntax: {label:} BSET Ws, bit4  
 [Ws],  
 [Ws]++,  
 [Ws]--,  
 [Ws++],  
 [Ws--],

Operands: bit4 ∈ [0 ... 15]; Ws ∈ [W0 ... W15]

Operation: 1 → Ws<bit4>

Status Affected: None

Encoding: 

1010	0000	bbbb	0000	0ppp	ssss
------	------	------	------	------	------

Description: Bit 'bit4' in register Ws is set.

The 'b' bits select value bit4 of the bit position to be cleared.

The 's' bits select the address of the source/destination register.

The 'p' bits select source address mode 2.

See Table 1-6 for modifier addressing information.

**Note:** This instruction operates in word mode only.

Words: 1

Cycles: 1

## Examples

Example1 BSET W6, #5 ; Set bit 5 in W6

Before Instruction

After Instruction

# BSETF

Bit Set f

Syntax: {label:} BSET.b f, bit3

Operands: bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]  
 Operation: 1 → f<bit3>  
 Status Affected: None  
 Encoding: 

1010	1000	bbbf	ffff	ffff	ffff
------	------	------	------	------	------

  
 Description: Bit 'bit3' in file register f is set.

The 'b' bits select value bit3 of the bit position to be cleared.  
 The 'f' bits select the address of the file register.

**Note:** This instruction operates in byte mode only.  
**Note:** The .b extension must be included with the opcode.

Words: 1  
 Cycles: 1

## Examples

Example1 BSET.B RAM135, #5 ; Set bit 5 in RAM135  
 Before Instruction  
 After Instruction



# BTFSC

Bit Test f, Skip if Clear

Syntax: {label:} BTSC.b f, bit3

Operands: bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]

Operation: Test (f)<bit3>, skip if clear

Status Affected: None

Encoding:	1010	1111	bbbf	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description: Bit 'bit3' in (f) is tested. If the bit is '0', then the fetched instruction is discarded and on the next cycle a NOP is executed instead.

The 'b' bits select the value bit3 of the bit position to be tested.  
The 'f' bits select the address of the file register.

- Note:** This instruction operates in byte mode only.
- Note:** The .b extension must be included with the opcode.

Words: 1  
Cycles: 1 (2 or 3)

## Examples

Example1 BTSC.b RAM135, #5 ; Bit test bit 5 in RAM135, skip if clear

Before Instruction

After Instruction



09876543210  
"01010101"  
10101010

BTFSS

Bit Test f, Skip if Set

Syntax: {label:} BTSS.b f, bit3

Operands: bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]  
Operation: Test (f)<bit3>, skip if set  
Status Affected: None

Encoding:	1010	1110	bbbf	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description: Bit 'bit3' in (f) is tested. If the bit is '1', then the fetched instruction is discarded and on the next cycle a NOP is executed instead.

The 'b' bits select value bit3 of the bit position to be cleared.  
The 'f' bits select the address of the file register.

**Note:** This instruction operates in byte mode only.  
**Note:** The .b extension must be included with the opcode.

Words: 1  
Cycles: 1 (2 or 3)

Examples

Example1 BTSS.b RAM135, #5 ; Bit test bit 5 in RAM135, skip if set  
Before Instruction  
  
After Instruction



09870457-060101

# BTGF

Bit Toggle f

Syntax: {label;} BTG.b f, bit3

Operands: bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]

Operation: (f)<bit3> → (f)<bit3>

Status Affected: None

Encoding: 

1010	1010	bbbf	ffff	ffff	ffff
------	------	------	------	------	------

Description: Bit 'bit3' in file register f is toggled.

The 'b' bits select value bit3 of the bit position to be cleared.  
The 'f' bits select the address of the file register.

**Note:** This instruction operates in byte mode only.  
**Note:** The .b extension must be included with the opcode.

Words: 1

Cycles: 1

## Examples

Example1 BTG.b RAM135, #5 ; Toggle bit 5 in RAM135

Before Instruction

After Instruction

# BTSC

## Bit Test Ws, Skip if Clear

Syntax:	{label:}	BTSC	Ws,	bit4
			[Ws],	
			[Ws]++,	
			[Ws]--,	
			[Ws++] ,	
			[Ws--] ,	

Operands:	bit4 ∈ [0 ... 15]; Ws ∈ [W0 ... W15]					
Operation:	Test (Ws)<bit4>, skip if clear.					
Status Affected:	None					
Encoding:	1010	0111	bbbb	0000	0ppp	ssss
Description:	Bit 'bit4' in (Ws) is tested. If the bit is '0', then the fetched instruction is dis-					

The 'b' bits select value bit4 of the bit position to be tested.  
The 's' bits select the address of the source register.  
The 'p' bits select source address mode 2 (values 0-4).

See Table 1-5 for modifier addressing information.

**Note:** This instruction operates in word mode only.

Words:	1
Cycles:	1 (2 or 3)

### Examples

Example1	BTSC	W6, #5	; Test bit 5 in W6, skip if clear
	Before Instruction		
	After Instruction		

# BTSS

Bit Test Ws, Skip if Set

Syntax:	{label:}	BTSS	Ws,	bit4
			[Ws],	
			[Ws]++,	
			[Ws]--,	
			[Ws++],	
			[Ws--],	

Operands:	bit4 ∈ [0 ... 15]; Ws ∈ [W0 ... W15]					
Operation:	Test (Ws)<bit4>, skip if set.					
Status Affected:	None					
Encoding:	1010	0110	bbbb	0000	0ppp	ssss
Description:	Bit 'bit4' in (Ws) is tested. If the bit is '1', then the fetched instruction is discarded and on the next cycle a NOP is executed instead.					

The 'b' bits select the value bit4 of the bit position to be tested.  
 The 's' bits select the address of the source register.  
 The 'p' bits select source address mode 2 (values 0-4).

See Table 1-5 for modifier addressing information.

**Note:** This instruction operates in word mode only.

Words:	1
Cycles:	1 (2 or 3)

## Examples

Example1	BTSS	W6, #5	; Test bit 5 in W6, skip if set
	Before Instruction		
	After Instruction		

050303

Syntax:	{label:}	BTST.C	Ws,	bit4
		BTST.Z	[Ws],	
			[Ws]++,	
			[Ws]--,	
			[Ws++] ,	
			[Ws--] ,	

Operation:           if “.Z” option,  $\overline{(Ws)}_{\langle \text{bit4} \rangle} \rightarrow Z$   
                           if “.C” option,  $(Ws)_{\langle \text{bit4} \rangle} \rightarrow C$

Encoding:	1010	0011	bbbb	Z000	0ppp	ssss
-----------	------	------	------	------	------	------

The 'b' bits select value bit4 of the bit position to be test/set.  
The 'Z' bit selects the Z or C flag bit as destination.  
The 's' bits select the address of the source register.  
The 'p' bits select source address mode 2.

**Note:** This instruction operates in word mode only.

Cycles: 1

**Example1**                      **BTST.C    W6,#5**                      ; Test bit 5 in W6 to the C flag

### After Instruction

# BTSTF

## Bit Test f

Syntax: {label:} BTST.b f, bit3

Operands: bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]

Operation: (f)<bit3> → Z

Status Affected: Z

Encoding: 

1010	1011	bbbf	ffff	ffff	ffff
------	------	------	------	------	------

Description: Bit 'bit3' in file register f is tested, the Zero Flag bit is set if it is zero and cleared otherwise. The file register contents are unchanged.

The 'b' bits select value bit3 of the bit position to be cleared.  
The 'f' bits select the address of the file register.

**Note:** This instruction operates in byte mode only.  
**Note:** The .b extension must be included with the opcode.

Words: 1  
Cycles: 1

### Examples

Example1 BTST.b RAM135, #5 ; Test bit 5 in RAM135

Before Instruction

After Instruction

# BTSTS

## Bit Test/Set in Ws

Syntax:	{label:}	BTSTS.C	Ws,	bit4
		BTSTS.Z	[Ws],	
			[Ws]++,	
			[Ws]--,	
			[Ws++] ,	
			[Ws--],	

Operands:	bit4 ∈ [0 ... 15]; Ws ∈ [W0 ... W15]				
Operation:	if “.Z” option, first $\overline{(Ws)}\langle bit4 \rangle \rightarrow Z$ , then $1 \rightarrow Ws\langle bit4 \rangle$ if “.C” option, first $(Ws)\langle bit4 \rangle \rightarrow C$ , then $1 \rightarrow Ws\langle bit4 \rangle$				
Status Affected:	C or Z				
Encoding:	1010	0100	bbbb	Z000	
Description:	Bit ‘bit4’ in register Ws is tested and then set.				

The 'b' bits select the value bit4 of the bit position to be test/set.  
The 'Z' bit selects the Z or C flag bit as destination.  
The 's' bits select the address of the source register.  
The 'p' bits select source address mode 2.

See Table 1-5 for modifier addressing information.

**Note:** This instruction operates in word mode only.

Words:	1
Cycles:	1

### Examples

Example1	BTSTS.Z W6,#5	; Test/Set bit 5 in W6 to the Z flag
	Before Instruction	
	After Instruction	



**BTSTSF**

### Bit Test/Set f

**Syntax:** {label:} BTSTS.b f, bit3

**Operands:** bit3  $\in [0 \dots 7]$ ; f  $\in [0 \dots 8191]$

Operation: First  $\overline{(f)\langle \text{bit3} \rangle} \rightarrow Z$ , then  $1 \rightarrow (f)\langle \text{bit3} \rangle$

Status Affected: Z

**Encoding:**

1010	1100	bbbf	ffff	ffff	ffff
------	------	------	------	------	------

<b>Description:</b>	Bit 'bit3' in file register f is tested and then set.
---------------------	---

The 'b' bits select value bit3 of the bit position to be cleared.

The 'f' bits select the address of the file register.

**Note:** This instruction operates in byte mode only.

**Note:** The .b extension must be included with the opcode.

Words: 1

Cycles: 1

## Examples

**Example1**                      BTSTS.b RAM135, #5                      ; Test/Set bit 5 in RAM135

### Before Instruction

### After Instruction

**09-06-2017**

BTSTW

Bit Test in Ws

Syntax:	{label:}	BTST.C	Ws,	Wb
		BTST.Z	[Ws],	
			[Ws]++,	
			[Ws]--,	
			[Ws++] ,	
			[Ws--],	

Operands: Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]

Operation: if ".Z" option, (Ws)<(Wb)> → Z  
if ".C" option, (Ws)<(Wb)> → C

Status Affected: C or Z

Encoding:	1010	0101	Zwww	w000	0ppp	ssss
-----------	------	------	------	------	------	------

Description: Bit (Wb) in register Ws is tested.  
The Zero flag contains the inversion of the bit or the Carry flag contains the bit.

The 'w' bits select the address of the bit select register.  
The 'Z' bit selects the Z or C flag bit as destination.  
The 's' bits select the address of the source register.  
The 'p' bits select source address mode 2.

See Table 1-5 for modifier addressing information.

Words: 1  
Cycles: 1

Examples

Example1 BTST.C W5,W6 ; Test bit in W5 selected by W6  
Before Instruction  
  
After Instruction

### Branch if Zero

**Syntax:** {label:} BRA BZ, Slit16

Operands:                    Slit16 ∈ [-32768 ... +32767]

Operation: Condition = Z  
if (condition), then  $(PC+2) + 2*Slit16 \rightarrow PC$ , and NOP  $\rightarrow$  Instruction Register.

Status Affected: None

Encoding:	0011	0010	nnnn	nnnn	nnnn	nnnn
-----------	------	------	------	------	------	------

Description:	If the Z Flag is '1', then the program will branch.
--------------	---

The 2's complement number '2\*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be  $(PC+2) + 2*Slit16$ . This instruction is then a two-cycle instruction, with a NOP in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2)

## Examples

**Example1**                      BRA      Z, label                      ; Branch if Zero

### Before Instruction

### After Instruction

**THE UNIVERSITY OF CHICAGO**

# CALL

## Call Subroutine

Syntax: {label;} CALL lit23  
CALL.S

Operands: lit23 ∈ [0 ... 8388606]

Operation: (PC) +4 → PC,  
(PC<15:0>) → TOS,  
(W15)+2 → W15  
(PC<23:16>) → TOS,  
(W15)+2 → W15  
lit23 → PC, NOP → Instruction Register.  
If S = 1, copy the contents of the primary registers into the shadow registers.

Status Affected: None

Encoding:

1st word

2nd word

Description:

0000	001S	nnnn	nnnn	nnnn	nnn0
0000	0000	0000	0000	0nnn	nnnn

Subroutine call of entire 4M instruction program memory range. First, return address (PC+4) is pushed onto the return stack (24-bits wide).

Then the 24-bit value 'lit23' is loaded into the PC. CALL is a two-cycle instruction.

The 'n' bits form the target address.

If 'S' = 1, the primary registers are copied into the shadow registers.

If 'S' = 0, no update occurs.

Words: 2

Cycles: 2

### Examples

Example1 CALL label ; Call subroutine

Before Instruction

After Instruction

# CALLW

Call Indirect Subroutine

Syntax: {label:} CALL Wn  
CALL.S

Operands: Wn ∈ [W0, W15]  
 Operation: (PC) +2 → PC,  
 (PC<15:0>) → TOS,  
 (W15)+2 → W15  
 (PC<23:16>) → TOS,  
 (W15)+2 → W15  
 0 → PC<22:17>, (Wn) → PC<16:1>, 0 → PC<0>;  
 NOP → Instruction Register.

Status Affected: None

Encoding:	0000	0001	S000	0000	0000	ssss
-----------	------	------	------	------	------	------

Description: Indirect subroutine call of first 64K instructions of program memory. First, return address (PC+2) is pushed onto the return stack.

Then, the 16-bit value (Wn) is left shifted 1 bit, zero-extended and loaded into the PC. CALL is a two-cycle instruction.

Words: 1  
 Cycles: 2

## Examples

Example1 CALL W5 ; Call indirect subroutine  
 Before Instruction  
 After Instruction

# CLR

## Clear Ws

Syntax:	{label:}	CLR{.b}	Ws
			[Ws]
			[Ws]++
			[Ws]--
			[Ws++]
			[Ws--]

Operands: Ws ∈ [W0 ... W15]  
Operation: 0 → Ws  
Status Affected: Z

Encoding:	1110	1011	0B00	0000	0ppp	ssss
-----------	------	------	------	------	------	------

Description: The contents of the source register are cleared and the Z flag is set.

The 'B' bits selects byte or word operation.  
The 's' bits select the address of the source register.  
The 'p' bits select the source address mode 2 (values 0-4).

See Table 1-5 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

### Examples

Example1 CLR W7 ; Clear

Before Instruction

After Instruction

09870457-060101  
TOT090-25404860

# CLRAC

Clear Accumulator, Prefetch Operands

Syntax:	{label:} CLR	A,	,Wxp,[Wx]	,Wyp,[Wy]	,AWB
		B,	,Wxp,[Wx]+=kx	,Wyp,[Wy]+=ky	none
			,Wxp,[Wx]-=kx ‡	,Wyp,[Wy]-=ky ‡	
			,Wxp,[W5+W8]	,Wyp,[W7+W8]	
			none	none	

‡ Alternate format for negative kx,ky

Operands: Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6};  
Wyp ∈ {W0 ... W3}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6};  
AWB ∈ {W9, [W9]++}

Operation: 0 → ACC(A or B)  
([Wx]) → Wxp; (Wx)+kx → Wx;  
([Wy]) → Wyp; (Wy)+ky → Wy;  
(ACC(B or A)) rounded → AWB

Status Affected: OA, OB, SA, SB

Encoding:	1100	0011	A0xx	yyii	ijjj	jjaa
-----------	------	------	------	------	------	------

Description: Clear the specified accumulator, prefetch operands and optionally store accumulator results in preparation for a repeated MAC type instruction. Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required. Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required. AWB specifies the direct or indirect store of the convergently rounded contents of other accumulator, if required. Note that the specification of (B or A) is consistent with the MAC instruction. For example, CLRAC A, W9 will store ACCB into W9.

The 'A' bit selects the other accumulator used for write back.

The 'i' bits select the Wx pre-fetch operation.

The 'j' bits select the Wy pre-fetch operation.

The 'x' bits select the pre-fetch Wxp destination.

The 'y' bits select the pre-fetch Wyp destination.

The 'a' bits select the accumulator write-back destination.

See Table 1-9 through Table 1-14 for modifier addressing information.

Words: 1

Cycles: 1

Examples

Example1

CLR     A,W0,[W4]-=6,W1,[W6],[W9]++

; Clear ACCA, prefetch, move ACCB  
to [W9]++

Before Instruction

After Instruction



# CLRF

Clear f or Ww

Syntax: {label:} CLR{.b} f  
Ww

Operands: f ∈ [0 ... 8191]  
Operation: 0 → destination designated by D  
Status Affected: Z  
Encoding: 

1110	1111	0BDf	ffff	ffff	ffff
------	------	------	------	------	------

  
Description: Clear the register designated by D: If the optional Ww is specified, D=0 and clear Ww; otherwise, D=1 and clear the file register. Z flag is set.  
  
The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.  
The 'D' bit selects the destination.  
  
**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 CLR 345 ; Clear file register 345  
Before Instruction  
  
After Instruction

# CLRWDT

Clear Watchdog Timer

Syntax: {label:} CLRWDT

Operands: none

Operation: 0 → WDT Reg

Status Affected:  $\overline{TO}$ ,  $\overline{PD}$

Encoding:	1111	1110	0110	0000	0000	0000
-----------	------	------	------	------	------	------

Description: Clear the WatchDog Timer register.

Words: 1

Cycles: 1

## Examples

Example1 CLRWDT ; Clear Watchdog Timer

Before Instruction

After Instruction

# COM

## Complement Ws

Syntax:	{label:}	COM{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++] ,	[Wd++]
			[Ws--],	[Wd--]

Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	$\overline{(Ws)} \rightarrow Wd$					
Status Affected:	Z, N					
Encoding:	1110	1010	1Bqq	qddd	dppp	ssss
Description:	Compute the 1's complement of the contents of the source register Ws and place the result in the destination register Wd.					

The 'B' bit selects byte or word operation.  
 The 's' bits select the address of the source register.  
 The 'd' bits select the address of the destination register.  
 The 'p' bits select the source address mode 2 (values 0-4).  
 The 'q' bits select the destination address mode 2 (values 0-4).

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:	1
Cycles:	1

### Examples

Example1	COM	W5,W7	; Complement
	Before Instruction		
	After Instruction		

# COMF

Complement f

Syntax: {label:} COM{.b} f {,Ww}

Operands:

Operation:

Status Affected:

Encoding:

Description:

f ∈ [0 ... 8191]

$\overline{(f)}$  → destination designated by D

Z, N

1110	1110	1BDf	ffff	ffff	ffff
------	------	------	------	------	------

Compute the 1's complement of the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.  
The 'D' bit selects the destination.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1

COMF    RAM135    ; Complement

Before Instruction

After Instruction

09870457-060101  
T0T090"25402860

CP

Compare Wb with Ws, Set status flags

Syntax:	{label:}	CP{.b}	Wb,	Ws
				[Ws]
				[Ws]++
				[Ws]--
				[Ws++]
				[Ws--]

Operands: Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]

Operation: (Ws) - (Wb)

Status Affected: C, DC, N, OV, Z

Encoding:	1110	0001	0www	wB00	0ppp	ssss
-----------	------	------	------	------	------	------

Description: Compute (Ws) - (Wb), equivalent to SUBR instruction, then set flags but do not store result.

The 'B' bit selects byte or word operation.  
The 'p' bits select source address mode 2.  
The 'w' bits select the address of the Wb source register.  
The 's' bits select the address of the Ws source register.

See Table 1-5 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

Examples

Example1 CP W5,W6 ; Skip  
Before Instruction  
  
After Instruction

CP0

Compare 0x0000 with Ws, Set status flags

Syntax:

{label:} CP0{.b} Ws

[Ws]

[Ws]++

[Ws]--

[Ws++]

[Ws--]

Operands:

Ws ∈ [W0 ... W15]

Operation:

(Ws) - 0x0000

Status Affected:

C, DC, N, OV, Z

Encoding:

1110	0000	0B00	0000	0ppp	ssss
------	------	------	------	------	------

Description:

Compute (Ws) - 0x0000, set flags but do not store result.

The 'B' bit selects byte or word operation.  
The 'p' bits select source address mode 2.  
The 's' bits select the address of the Ws source register.

See Table 1-5 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:

1

Cycles:

1

Examples

Example1

CP0 W5 ; Compare

Before Instruction

After Instruction

**UNIVERSITY OF CALIFORNIA**

Syntax:	{label:}	CP1{.b}	Ws
			[Ws]
			[Ws]++
			[Ws]--
			[Ws++]
			[Ws--]

<b>Description:</b>	Compute (Ws) - 0xFFFF, set flags but do not store result.
---------------------	---

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Cycles: 1

### After Instruction

CPB

Compare Wb with Ws with Borrow, set status flags

Syntax:	{label:}	CPB{.b}	Wb,	Ws
				[Ws]
				[Ws]++
				[Ws]--
				[Ws++]
				[Ws--]

Operands: Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]

Operation: (Ws) - (Wb) - (C̄)

Status Affected: C, DC, N, OV, Z

Encoding:	1110	0001	1www	wB00	0ppp	ssss
-----------	------	------	------	------	------	------

Description: Compute (Ws) - (Wb) - (c), equivalent to SUBRB instruction, then set flags but do not store result.

The 'B' bit selects byte or word operation.  
The 'p' bits select source address mode 2.  
The 'w' bits select the address of the Wb source register.  
The 's' bits select the address of the Ws source register.

See Table 1-5 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

Examples

Example1 CPB W5,W6 ; Skip  
Before Instruction  
  
After Instruction



# CPBLS

Compare Wb with lit5 with borrow, Set status flags

Syntax: {label:} CPB{.b} Wb, lit5

Operands: Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]

Operation: (Wb) - lit5 - (C̄)

Status Affected: C, DC, N, OV, Z

Encoding:	1110	0001	1www	wB00	011k	kkkk
-----------	------	------	------	------	------	------

Description: Compute (Wb) - lit5, set flags but do not store result.

The 'B' bit selects byte or word operation.  
The 'w' bits select the address of the Wb source register.  
The 'k' bits provide the literal operand, a five bit integer number.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

Example1 CPB W5, #30

Before Instruction

After Instruction

# CPF

Compare f with Ww, Set status flags

Syntax: {label;} CP{.b} f

Operands: f ∈ [0 ...8191]

Operation: (f) - (Ww)

Status Affected: C, DC, N, OV, Z

Encoding: 

1110	0011	0B0f	ffff	ffff	ffff
------	------	------	------	------	------

Description: Compute (f) - (Wd), set flags but do not store result.

The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 CP RAM135 ; Compare  
Before Instruction  
  
After Instruction

# CPF0

Compare f with 0x0000, Set status flags

Syntax: {label;} CP0{.b} f

Operands: f ∈ [0 ... 8191]

Operation: (f) - 0x0000

Status Affected: C, DC, N, OV, Z

Encoding: 

1110	0010	0B0f	ffff	ffff	ffff
------	------	------	------	------	------

Description: Compute (f) - 0x0000, set flags but do not store result.

The 'B' bit selects byte or word operation.

The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

Example1 CP0 53 ; Compare

Before Instruction

After Instruction

CPF1

Compare f with 0xFFFF, Skip if Equal (f = 0FFFFh)

Syntax:

{label:} CPF1{.b} f

Operands:

f ∈ [0 ... 8191]

Operation:

(f) - 0xFFFF

Status Affected:

C, DC, N, OV, Z

Encoding:

1110	0010	1B0f	ffff	ffff	ffff
------	------	------	------	------	------

Description:

Compute (f) - 0xFFFF, set flags but do not store result.

The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:

1

Cycles:

1

Examples

Example1

CP1 53 ; Compare

Before Instruction

After Instruction

# CPFB

Compare f with Ww with Borrow, Set status flags

Syntax: {label:} CPB{.b} f

Operands:	f ∈ [0 ...8191]					
Operation:	(f) - (Ww) - ( $\overline{C}$ )					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1110	0011	1B0f	ffff	ffff	ffff
Description:	Compute (f) - (Ww) - ( $\overline{C}$ ), set flags but do not store result.					

The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:	1
Cycles:	1

## Examples

Example1	CPB	RAM135	; Compare RAM135-Ww
	Before Instruction		
	After Instruction		

# CPFSEQ Compare f with Ww, Skip if Equal (f = Ww)

Syntax: {label:} CPFSEQ{.b} f

Operands:

f ∈ [0 ... 8191]

Operation:

(f) - (Ww)  
 Skip if (f) = (Ww)

Status Affected:

None

Encoding:

1110	0111	1B0f	ffff	ffff	ffff
------	------	------	------	------	------

Description:

Compares the contents of data memory location 'f' to the contents of working register Ww by performing a subtraction.

If (f) = (Ww) then the fetched instruction is discarded and on the next cycle a NOP is executed instead.

The 'B' bit selects byte or word operation.

The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:

1

Cycles:

1 (2 or 3)

## Examples

Example1

CPFSEQ RAM135 ; Compare

Before Instruction

After Instruction

09870457,060101  
T0T090"/5402860

CPFSGT

Signed Compare f with Ww, Skip if Greater Than (f >Ww)

Syntax: {label:} CPFSGT{.b} f

Operands: f ∈ [0 ... 8191]  
Operation: (f) - (Wd)  
Skip if (f) > (Wd)

Status Affected: None

Encoding:	1110	0110	0B0f	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description: Compares the contents of data memory location 'f' to the contents of working register Ww by performing a subtraction.  
If (f) > (Ww) then the fetched instruction is discarded and on the next cycle a NOP is executed instead.

The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1 (2 or 3)

Examples

Example1 CPFSGT RAM135 ; Compare  
Before Instruction  
  
After Instruction

CPFSLT

Signed Compare f with Ww, Skip if Less Than (f < Ww)

Syntax:

{label:} CPFSLT{.b} f

Operands:

f ∈ [0 ... 8191]

Operation:

(f) - (Ww)  
Skip if (f) < (Ww)

Status Affected:

None

Encoding:

1110	0110	1B0f	ffff	ffff	ffff
------	------	------	------	------	------

Description:

Compares the contents of data memory location 'f' to the contents of working register Ww by performing a subtraction.

If (f) < (Ww) then the fetched instruction is discarded and on the next cycle a NOP is executed instead.

The 'B' bit selects byte or word operation.

The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:

1

Cycles:

1 (2 or 3)

Examples

Example1

CPFSLT RAM135 ; Compare

Before Instruction

After Instruction



CPFSNE

Signed Compare f with Ww, Skip if not Equal (f ≠ Ww)

Syntax:

{label:} CPFSNE{.b} f

Operands:

f ∈ [0 ...8191]

Operation:

(f) - (Ww)  
Skip if (f) ≠ (Ww)

Status Affected:

None

Encoding:

1110	0111	0B0f	ffff	ffff	ffff
------	------	------	------	------	------

Description:

Compares the contents of data memory location 'f' to the contents of working register Ww by performing a subtraction.  
If (f) ≠ (Ww) then the fetched instruction is discarded and on the next cycle a NOP is executed instead.

The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.

Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:

1

Cycles:

1 (2 or 3)

Examples

Example1

CPFSNE RAM135 ; Compare

Before Instruction

After Instruction

# CPLS

## Compare Wb with lit5, Set status flags

Syntax: {label:} CP{.b} Wb, lit5

Operands: Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]

Operation: (Wb) - lit5

Status Affected: C, DC, N, OV, Z

Encoding: 

1110	0001	0www	wB00	011k	kkkk
------	------	------	------	------	------

Description: Compute (Wb) - lit5, set flags but do not store result.

The 'B' bit selects byte or word operation.

The 'w' bits select the address of the Wb base register.

The 'k' bits provide the literal operand, a five bit integer number.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

### Examples

Example1 CP W5, #30

Before Instruction

After Instruction

FOR90" 25102860

# DAW

## Decimal Adjust Wn

Syntax: {label:} DAW.b Wn

Operands: Wn ∈ [W0 ... W15]  
 Operation: If [Wn<3:0> >9] or [DC = 1] then  
               (Wn<3:0> + 6 → Wn<3:0>  
               else  
               (Wn<3:0>) → Wn<3:0>;  
               If [Wn<7:4> >9] or [C = 1] then  
               (Wn<7:4>) + 6 → Wn<7:4>  
               else  
               (Wn<7:4>) → Wn<7:4>;

Status Affected:

C

Encoding:

1111	1101	0100	0000	0000	ssss
------	------	------	------	------	------

Description:

DAW adjusts the eight bit value in Wn (LSB's) resulting from the earlier addition of two variables (each in packed BCD format) and produces a correct packed BCD result.

The 's' bits select the address of the source register.

- Note:** This instruction operates in byte mode only.
- Note:** The .b extension must be included with the opcode.

Words: 1  
 Cycles: 1

### Examples

Example1                                DAW.b    W5                                ; Decimal adjust

Before Instruction

After Instruction

# DEC

### Decrement Ws

Syntax:	{label:}	DEC{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++] ,	[Wd++]
			[Ws--] ,	[Wd--]

Operands:  $Ws \in [W0 \dots W15]; Wd \in [W0 \dots W15]$

Operation:  $(Ws) - 1 \rightarrow Wd$

**Status Affected:** C, DC, N, OV, Z

Encoding:	1110	1001	0Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description:	Subtract one from the contents of the source register Ws and place the result in the destination register Wd.
--------------	---

The 'B' bit selects byte or word operation.

The 's' bits select the address of the source register.

The 'd' bits select the address of the destination register.

The 'p' bits select the source address mode 2 (values 0-4).

The 'q' bits select the destination address mode 2 (values 0-4).

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

Example1	DEC	W5,W7	; Decrement
----------	-----	-------	-------------

### Before Instruction

### After Instruction

# DEC2

## Decrement Ws by 2

Syntax:	{label:}	DEC2{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]

Operands: Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]

Operation: (Ws) - 2 → Wd

Status Affected: C, DC, N, OV, Z

Encoding:	1110	1001	1Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Subtract two from the contents of the source register Ws and place the result in the destination register Wd.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select the source address mode 2 (values 0-4).
- The 'q' bits select the destination address mode 2 (values 0-4).

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

### Examples

Example1	DEC2	W5,W7	; Decrement
	Before Instruction		
	After Instruction		

# DECF

Decrement f

Syntax: {label:} DEC{.b} f {,Ww}

Operands: f ∈ [0 ... 8191]  
Operation: (f) - 1 → destination designated by D  
Status Affected: C, DC, N, OV, Z

Encoding:	1110	1101	0BDf	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description: Subtract one from the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.  
The 'D' bit selects the destination.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1                      DECF    RAM135                      ; Decrement

Before Instruction

After Instruction

09370457-050104  
T0T090-25402860

# DECFSNZ

Decrement f, Skip if Not Zero

Syntax: {label:} DECSNZ{.b} f {,Ww}

Operands: f ∈ [0 ... 8191]  
Operation: (f) - 1 → destination designated by D; skip if result ≠ 0  
Status Affected: None  
Encoding: 

1110	0101	1BDf	ffff	ffff	ffff
------	------	------	------	------	------

  
Description: Subtract one from the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register. If the result ≠ 0, then the fetched instruction is discarded and on the next cycle a NOP is executed instead.  
  
The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination  
The 'f' bits select the address of the file register.  
  
**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1 (2 or 3)

## Examples

Example1 DECSNZ RAM135, Ww ; Decrement  
Before Instruction  
  
After Instruction

Syntax: {label:} DECSZ{.b} f {,Ww}

Encoding:	1110	0101	0BDF	ffff	ffff	ffff
-----------	------	------	------	------	------	------

### After Instruction



DISI

Disable Interrupts

Syntax:

{label:}

DISI

lit14

Operands:

lit14 ∈ [0 ... 16384]

Operation:

Disable interrupts for lit14 cycles

Status Affected:

None

Encoding:

1111	1100	00kk	kkkk	kkkk	kkkk
------	------	------	------	------	------

Description:

This instruction disables the interrupts for lit14 instruction cycles after the instruction executes. This instruction can be used before critical code sections to ensure un-interrupted execution.

Words:

1

Cycles:

1

Examples

Example1

DISI

#30

; Disable interrupts for next 30 instruction cycles

Before Instruction

After Instruction

DIV

Divide TBD

Syntax: {label:} DIV TBD

Operands: TBD  
Operation: TBD  
Status Affected: TBD  
Encoding: 

1101	100x	xxxx	xxxx	xxxx	xxxx
------	------	------	------	------	------

  
Description: TBD

**Note:** Word operation is assumed.

Words: 1  
Cycles: TBD

Examples

Example1                      DIV      TBD                      ; Divide  
Before Instruction  
  
After Instruction

DO

Initialize Hardware loop

Syntax:

{label:}

DO

Slit16,

lit14

Operands:

Slit16 ∈ [-32768 ... +32767];  
lit14 ∈ [0 ... 16383]

Operation:

Push Shadows  
 (lit14) → DOCOUNT (Loop Count Register)  
 (PC)+4 → PC  
 (PC) → DOSTART (Loop Start Register)  
 (PC) + (2\*Slit16) → DOEND (Loop End Register)  
 Enable Code Looping

Status Affected:

None

Encoding:

0000	1000	00kk	kkkk	kkkk	kkkk
0000	0000	nnnn	nnnn	nnnn	nnnn

Description:

Repeat lit14 times the code segment delineated by the address of the instruction immediately following the DO instruction and an end address formed by the address of the first instruction plus offset Slit16.  
  
 The 'k' bits specify the loop count.  
 The 'n' bits are a signed literal that specifies the number of instructions off-set from (PC+4) of the last instruction executed in the loop.  
  
**Note 1:** The value k = 0 is invalid.  
**2:** The value n=-1 is invalid. The DO instruction is not allowed to generate a DO loop only including itself.  
**3:** n=0 will generate a loop size of 1 word (same as REPEAT instruction except instruction is fetched every iteration).

Words:

2

Cycles:

2 + n\*(# of cycles required to execute loop)

Examples

Example1

DO        #5, #6                ; Do next 5 instructions 6 times

Before Instruction

After Instruction

DOW

Initialize Hardware loop

Syntax:

{label:} DO Slit16, Wn

Operands:

Slit16 ∈ [-32768 ... +32767];  
 Wn ∈ [W0 ... W15]

Operation:

Push Shadows  
 (Wn) → DOCOUNT (Loop Count Register)  
 (PC)+4 → PC  
 (PC) → DOSTART (Loop Start Register)  
 (PC) + (2\*Slit16) → DOEND (Loop End Register)  
 Enable Code Looping

Status Affected:

None

Encoding:

0000	1000	1000	0000	0000	ssss
0000	0000	nnnn	nnnn	nnnn	nnnn

Description:

Repeat (Wn) times the code segment delineated by the address of the instruction immediately following the DO instruction and an end address formed by the address of the first instruction plus offset Slit16.

The 's' bits specify the register Wn that contains the loop count (only the 14 LSBs of (Wn) are considered).

The 'n' bits are a signed literal that specifies the number of instructions off-set from (PC+4) of the last instruction executed in the loop.

**Note 1:** The value (Wn) = 0 is invalid.

**2:** The value n=-1 is invalid. The DO instruction is not allowed to generate a DO loop only including itself.

**3:** n=0 will generate a loop size of 1 word (same as REPEAT instruction except instruction is fetched every iteration).

Words:

2

Cycles:

2 + n\*(# of cycles required to execute loop)

Examples

Example1

DO #5,W6 ; Do next 5 instructions (W6) times

Before Instruction

After Instruction

TOP SECRET 060457Z090900

ED

Euclidean Distance

Syntax:	{label:} ED	A, Wm*Wm	,Wxp,[Wx]	,[Wy]
		B,	,Wxp,[Wx]+=kx	,[Wy]+=ky
			,Wxp,[Wx]-=kx ‡	,[Wy]-=ky ‡
			,Wxp,[W5+W8]	,[W7+W8]
			none	none

‡ Alternate format for negative kx,ky

Operands: Wm\*Wm ∈ {W0\*W0; W1\*W1; W2\*W2; W3\*W3}  
Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6};  
Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6};

Operation: (Wm)\*(Wm) → ACC(A or B);  
([Wx]-[Wy])→ Wxp; (Wx)+kx→Wx; (Wy)+ky→Wy;

Status Affected: OA, OB, SA, SB

Encoding:	1111	00mm	A1xx	00ii	ijjj	jj11
-----------	------	------	------	------	------	------

Description: Instruction to compute (A-B)<sup>2</sup> functions. Prefetch computes difference of prefetched values. Then, the Wm register is squared. The 32-bit result is sign-extended to 40-bits and written to the specified accumulator.  
Wx register specifies the prefetch of the minuend register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.  
Wy register specifies the prefetch of the subtrahend register. Post-modify Wy as required.  
Wxp contains the difference result.

The 'm' bits select the operand register Wm for the square:  
The 'A' bit selects the accumulator for the result.  
The 'i' bits select the Wx pre-fetch operation.  
The 'j' bits select the Wy pre-fetch operation.  
The 'x' bits select the pre-fetch difference Wxp destination.

See Table 1-9 through Table 1-14 for modifier addressing information.

Words: 1  
Cycles: 1

## Examples

Example1    ED    A, W2\*W2, W0, [W4]-=6, [W6]    ; Euclidean Distance to ACCA

Before Instruction

ACCA = 2

ACCB = 3

W0 = 5

W1 = 6

W2 = 7

W3 = 8

W8 = 1000

W10 = 2000

RAM(994) = 16

RAM(1000) = 17

RAM(2000) = 18

After Instruction

ACCA =  $2 + 7 * 8 = 58$

ACCB = 3

W0 = 17

W1 = 18

W2 = 7

W3 = 8

W8 = 994

W10 = 2000

RAM(994) = 3

RAM(1000) = 17

RAM(2000) = 18

FOR090" 25402860

# EDAC

## Square and Accumulate

Syntax:	{label:} EDAC	A, Wm*Wm	,Wxp,[Wx]	,Wyp,[Wy]	,AWB
	B,		,Wxp,[Wx]+=kx	,Wyp,[Wy]+=ky	<i>none</i>
			,Wxp,[Wx]-=kx ‡	,Wyp,[Wy]-=ky ‡	
			,Wxp,[W5+W8]	,Wyp,[W7+W8]	
			<i>none</i>	<i>none</i>	

‡ Alternate format for negative kx,ky

Operands: Wm\*Wm ∈ {W0\*W0; W1\*W1; W2\*W2; W3\*W3}  
Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6};  
Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6};  
AWB ∈ {W9, [W9]++}

Operation: (ACC(A or B)) + (Wm)\*(Wm) → ACC(A or B);  
([Wx]-[Wy]) → Wxp; (Wx)+kx → Wx; (Wy)+ky → Wy;  
(ACC(B or A)) rounded → AWB

Status Affected: OA, OB, SA, SB

Encoding:	1111	00mm	A1xx	00ii	ijjj	jjaa
-----------	------	------	------	------	------	------

Description: Instruction to compute  $(A-B)^2$  functions. Prefetch computes difference of prefetched values. Then, the Wm register is squared. The 32-bit result is sign-extended to 40-bits and added to the specified accumulator.  
Wx register specifies the prefetch of the minuend register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.  
Wy register specifies the prefetch of the subtrahend register. Post-modify Wy as required.  
Wxp contains the difference result.

The 'm' bits select the operand register Wm for the square:  
The 'A' bit selects the accumulator for the result. The other accumulator is used for write back.  
The 'i' bits select the Wx pre-fetch operation.  
The 'j' bits select the Wy pre-fetch operation.  
The 'x' bits select the pre-fetch difference Wxp destination.  
The 'a' bits select the accumulator write-back destination.

See Table 1-9 through Table 1-14 for modifier addressing information.

Words: 1  
Cycles: 1

## Examples

Example1    EDAC    A,W2\*W2,W0,[W4]-=6,W1,[W6],[W9]++ ; Square and Accumulate A

Before Instruction

ACCA = 2

ACCB = 3

W0 = 5

W1 = 6

W2 = 7

W3 = 8

W8 = 1000

W10 = 2000

RAM(994) = 16

RAM(1000) = 17

RAM(2000) = 18

After Instruction

ACCA =  $2+7*8=58$

ACCB = 3

W0 = 17

W1 = 18

W2 = 7

W3 = 8

W8 = 994

W10 = 2000

RAM(994) = 3

RAM(1000) = 17

RAM(2000) = 18



# EXCH

## Exchange Ws and Wd

Syntax: {label:} EXCH Wns, Wnd

Operands: Wns  $\in$  [W0 ... W15]; Wnd  $\in$  [W0 ... W15]

Operation: (Wns)  $\leftrightarrow$  (Wnd)

Status Affected: None

Encoding: 

1111	1101	0000	0ddd	d000	ssss
------	------	------	------	------	------

Description: This instruction exchanges the contents of two working registers.

The 's' bits select the address of one of the registers.

The 'd' bits select the address of the other register.

**Note:** Word operation is assumed.

Words: 1

Cycles: 1

### Examples

Example1 EXCH W5,W6 ; Exchange W5 and W6

Before Instruction

After Instruction

09870457-060101  
TOT090-4540960

**SECRET**

Syntax:	{label:}	FBCL{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++] ,	[Wd++]
			[Ws--] ,	[Wd--]

Operation:	See description
------------	-----------------

**Z**

1101	1111	1Bqq	qddd	dppp	ssss
------	------	------	------	------	------

Finds the first occurrence of a one (for a positive signed value) or zero (for a negative signed value) starting from the most significant bit after the sign bit working towards the least significant bit of the byte or word operand. The bit number will be placed in the destination effective address.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select address mode 1 (values 0-4).
- The 'q' bits select address mode 2 (values 0-4).

**Note:** The extension {.b} in the instruction denotes a byte operation rather than a word operation. You may use a [.w] extension to denote a word operation, but it is not required.

Cycles: 1

Example1                      FBCL      W5, W6                      ; Find first not sign

Before Instruction

After Instruction

# FBCR

## Find First Bit Change from Right

Syntax:	{label:}	FBCR{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]

Operands: Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]

Operation: See description

Status Affected: Z

Encoding:	1101	1111	0Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Finds the first occurrence of a bit different from bit<0> starting from bit<1> working towards the most significant bit of the byte or word operand. The bit number will be placed in the destination effective address.

A result of zero (Z=1) indicates the bit was not found.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select address mode 1 (values 0-4).
- The 'q' bits select address mode 2 (values 0-4).

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

**Note:** The result from a FBCR of the memory mapped Accumulator overflow byte can be directly used as the (signed) operand for a SFTA/B instruction to scale (shift right) the accumulator contents.

### Examples

Example1	FBCR	W5, W6	; Find first not sign
	Before Instruction		
	After Instruction		

# FF0L

## Find First Zero from Left

Syntax:	{label:}	FF0L{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++] ,	[Wd++]
			[Ws--] ,	[Wd--]

Operands: Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]  
 Operation: See description  
 Status Affected: Z

Encoding:	1100	1110	1Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Finds the first occurrence of a zero starting from the most significant bit working towards the least significant bit of the byte or word operand. The bit number will be placed in the destination effective address.

The least significant bit is allocated number 1, the most significant number 8 (for byte operations) or 16 (for word operations). A result of zero (Z=1) indicates the bit was not found.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select address mode 1 (values 0-4).
- The 'q' bits select address mode 2 (values 0-4).

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension {.b} in the instruction denotes a byte operation rather than a word operation. You may use a [.w] extension to denote a word operation, but it is not required.

Words: 1  
 Cycles: 1

### Examples

Example1                      FF0L      W5, W6                      ; Find first zero

Before Instruction

After Instruction

FF0R

Find First Zero from Right

Syntax:	{label:}	FF0R{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]

Operands: Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]

Operation: See description

Status Affected: Z

Encoding:	1100	1110	0Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Finds the first occurrence of a zero starting from the least significant bit working towards the most significant bit of the byte or word operand. The bit number will be placed in the destination effective address.

The least significant bit is allocated number 1, the most significant number 8 (for byte operations) or 16 (for word operations). A result of zero (Z=1) indicates the bit was not found.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select address mode 1 (values 0-4).
- The 'q' bits select address mode 2 (values 0-4).

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension {.b} in the instruction denotes a byte operation rather than a word operation. You may use a [.w] extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

Examples

Example1	FF0R	W5, W6	; Find first zero
Before Instruction			
After Instruction			

# FF1L Find First One from Left

Syntax:	{label:}	FF1L{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]

Operands: Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]  
 Operation: See description  
 Status Affected: Z

Encoding:	1100	1111	1Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Finds the first occurrence of a one starting from the most significant bit working towards the least significant bit of the byte or word operand. The bit number will be placed in the destination effective address.

The least significant bit is allocated number 1, the most significant number 8 (for byte operations) or 16 (for word operations). A result of zero (Z=1) indicates the bit was not found.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select address mode 1 (values 0-4).
- The 'q' bits select address mode 2 (values 0-4).

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension {.b} in the instruction denotes a byte operation rather than a word operation. You may use a [.w] extension to denote a word operation, but it is not required.

Words: 1  
 Cycles: 1

## Examples

Example1 FF1L W5, W6 ; Find first one

Before Instruction

After Instruction

# FF1R

## Find First One from Right

Syntax:	{label:}	FF1R{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]

Operands: Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]

Operation: See description

Status Affected: Z

Encoding:	1100	1111	0Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Finds the first occurrence of a one starting from the least significant bit working towards the most significant bit of the byte or word operand. The bit number will be placed in the destination effective address.

The least significant bit is allocated number 1, the most significant number 8 (for byte operations) or 16 (for word operations). A result of zero (Z=1) indicates the bit was not found.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select address mode 1 (values 0-4).
- The 'q' bits select address mode 2 (values 0-4).

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension {.b} in the instruction denotes a byte operation rather than a word operation. You may use a [.w] extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

### Examples

Example1	FF1R	W5, W6	; Find first one
	Before Instruction		
	After Instruction		

**GOTO**

## Unconditional Branch

Syntax:	{label:}	GOTO	lit23
---------	----------	------	-------

Operands:  $\text{lit23} \in [0 \dots 8388606]$

**Operation:** lit23 → PC, NOP → Instruction Register.

Status Affected: None

**Encoding:**

1st word	0000	0100	nnnn	nnnn	nnnn	nnn0
2nd word	0000	0000	0000	0000	0nnn	nnnn

Description:	Unconditional branch to anywhere within the 4M instruction program memory range. GOTO is always a two-cycle instruction.
--------------	--

The 'n' bits form the target address.

**Words:** 2

Cycles: 2

## Examples

**Example1**                      GOTO    label                      ; Goto location at label

### Before Instruction

### After Instruction

**THE UNIVERSITY OF CHICAGO**



# GOTOW

## Unconditional Indirect Branch

Syntax: {label;} GOTO Wn

Operands: Wn ∈ [W0 ... W15]  
Operation: 0 → PC<22:17>, (Wn) → PC<16:1>, 0 → PC<0>;  
NOP → Instruction Register.  
Status Affected: None  
Encoding: 

0000	0001	0100	0000	0000	ssss
------	------	------	------	------	------

  
Description: Unconditional indirect branch within the first 64K instructions program memory range. GOTO is always a two-cycle instruction.

The 16-bit value (Wn) is left shifted 1 bit, zero-extended and loaded into the PC. CALL is a two-cycle instruction.

The 's' bits select the address of the source register.

Words: 1  
Cycles: 2

### Examples

Example1 GOTO W5 ; Goto location specified by contents of W5  
Before Instruction  
After Instruction

# HALT

## Halt

**Syntax:** {label:} HALT

Operands: none

Operation: No Operation, HALT

Status Affected: None

Encoding:	1111	1110	0010	0000	0000	0000
-----------	------	------	------	------	------	------

Description:	Stop the processor in an emulation environment.
--------------	---

Words: 1

Cycles: 1

## Examples

**Example1**                      **HALT**                      ; Halt

Before Instruction

### After Instruction

**SECRET**

# INC

## Increment Ws

Syntax:	{label:}	INC{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]

Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	(Ws) + 1 → Wd					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1110	1000	0Bqq	qddd	dppp	ssss
Description:	Add one to the contents of the source register Ws and place the result in the destination register Wd.					

The 'B' bit selects byte or word operation.  
 The 's' bits select the address of the source register.  
 The 'd' bits select the address of the destination register.  
 The 'p' bits select the source address mode 2 (values 0-4).  
 The 'q' bits select the destination address mode 2 (values 0-4).

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:	1
Cycles:	1

### Examples

Example1	INC	W5,W7	; Increment
	Before Instruction		
	After Instruction		

# THE NEW YORK PUBLIC LIBRARY

Syntax:	{label:}	INC2	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++] ,	[Wd++]
			[Ws--] ,	[Wd--]

Status Affected: C, DC, N, OV, Z

Encoding:	1110	1000	1Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

The 'q' bits select the destination address mode 2 (values 0-4).

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Cycles: 1

## Examples

### After Instruction

# INCf

Increment f

Syntax: {label:} INC{.b} f {,Ww}

Operands:
Operation:
Status Affected:
Encoding:
Description:

$f \in [0 \dots 8191]$

$(f) + 1 \rightarrow$  destination designated by D

C, DC, N, OV, Z

1110	1100	0Bdf	ffff	ffff	ffff
------	------	------	------	------	------

Add one to the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.  
The 'D' bit selects the destination.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1
Cycles: 1

## Examples

Example1

INC
RAM135
; Increment

Before Instruction

After Instruction

# INCFSNZ

Increment f, Skip if Not Zero

Syntax: {label:} INCSNZ{.b} f {,Ww}

Operands: f ∈ [0 ...8191]

Operation: (f) + 1 → destination designated by D; skip if result ≠ 0

Status Affected: None

Encoding:

1110	0100	1BDf	ffff	ffff	ffff
------	------	------	------	------	------

Description: Add one to the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register. If the result ≠ 0, then the fetched instruction is discarded and on the next cycle a NOP is executed instead.

The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1 (2 or 3)

## Examples

Example1 INCSNZ RAM135, Ww ; Increment

Before Instruction

After Instruction

# INCFSZ

Increment f, Skip if Zero

Syntax: {label:} INCSZ{.b} f {,Ww}

Operands:

Operation:

Status Affected:

Encoding:

Description:

f ∈ [0 ... 8191]

(f) + 1 → destination designated by D; skip if result = 0

None

1110	0100	0BDf	ffff	ffff	ffff
------	------	------	------	------	------

Subtract one from the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register. If the result = 0, then the fetched instruction is discarded and on the next cycle a NOP is executed instead.

The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:

Cycles:

1

1 (2 or 3)

## Examples

Example1

INCSZ    RAM135, Ww    ; Increment

Before Instruction

After Instruction

# IOR

Inclusive Or Wb and Ws

Syntax:	{label:}	IOR{.b}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws]++,	[Wd]++
				[Ws]--,	[Wd]--
				[Ws++],	[Wd++]
				[Ws--],	[Wd--]

Operands: Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]

Operation: (Wb).IOR.(Ws) → Wd

Status Affected: N, Z

Encoding:	0111	0www	wBqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Ior the contents of the source register Ws and the contents of the base register Wb and place the result in the destination register Wd.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'w' bits select the address of the base register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select source address mode 2.
- The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 IOR W5,W6,W7 ; Inclusive Or  
Before Instruction  
  
After Instruction



# IORLS

Inclusive Or Wb and Short Literal

Syntax:	{label:}	IOR{.b}	Wb	lit5	Wd
					[Wd]
					[Wd]++
					[Wd]--
					[Wd++]
					[Wd--]

Operands: Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]  
Operation: (Wb).IOR.lit5 → Wd  
Status Affected: N, Z

Encoding:	0111	0www	wBqq	qddd	d11k	kkkk
-----------	------	------	------	------	------	------

Description: Compute the Inclusive Or of the contents of the base register Wb and the literal operand and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.  
The 'w' bits select the address of the base register.  
The 'k' bits provide the literal operand, a five-bit integer number.  
The 'd' bits select the address of the destination register.  
The 'q' bits select destination address mode 2.

See Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1                    IOR    W5,#12,W7           ; Add  
Before Instruction  
  
After Instruction

# IORLW

Inclusive Or Literal and Wn

Syntax: {label:} IOR{.b} Slit10, Wn

Operands: Slit10 ∈ [-512 ... 511]; Wn ∈ [W0 ... W15]  
Operation: Slit10.IOR.(Wn) → Wn  
Status Affected: N, Z

Encoding: 

1011	0011	0Bkk	kkkk	kkkk	dddd
------	------	------	------	------	------

Description: Compute the Inclusive Or of the literal operand and the contents of the working register Wn and place the result in the working register Wn.

The 'B' bit selects byte or word operation.  
The 'd' bits select the address of the working register.  
The 'k' bits specify the literal operand, a signed 10-bit number.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 IOR #123,W7 ; Inclusive Or  
Before Instruction  
  
After Instruction

09070457-060404  
T0T090-25402860

# IORWF

Inclusive Or f and Ww

{label:} IOR{.b} f {,Ww}

Operands: f ∈ [0 ... 8191]  
Operation: (f).IOR.(Ww) → destination designated by D  
Status Affected: N, Z

Encoding:	1011	0111	0Bdf	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description: Compute the IOR of the contents of the working register and the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination.  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 IOR RAM135, Ww ; Inclusive Or  
Before Instruction  
  
After Instruction

09870457-060101

# ITCH

## Pop Shadow Registers

Syntax: {label;} POP.S

Operands: None  
Operation: Pop shadow registers  
Status Affected: All  
Encoding:  
Description: The values in the shadow registers are copied into the primary registers.  
Words: 1  
Cycles: 1

1111	1110	1000	0000	0000	0000
------	------	------	------	------	------

### Examples

Example1                      ITCH                      ; Itch  
Before Instruction  
  
After Instruction

# LAC

## Load Accumulator A

Syntax:	{label:}	LAC	A,	Wns,	[, Slit4]
			B,	[Wns],	
				[Wns]++	
				[Wns]--	
				[Wns--],	
				[Wns+Wb],	
				[Wns+lit5]	

Operands:	Wns ∈ [W0 ... W15]; Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31] Slit4 ∈ [-8 ... +7]					
Operation:	Shift <sub>Slit4</sub> (Extend(Wns)) → ACC					
Status Affected:	None					
Encoding:	1100	1010	Awww	wrrr	rggg	ssss
Description:	Read the contents of the effective address. Optionally shift, then place result in accumulator.					

The value contained at the effective address is assumed to be Q15 fractional data and is automatically sign-extended (through bit 39) and zero-backfilled (bits [15:0]) prior to shifting.

The 'A' bits specify the destination accumulator.  
The 's' bits specify the source register Wns.  
The 'g' bits select source address mode 3.  
The 'w' bits specify the offset amount lit5 OR the offset register Wb.  
The 'r' bits encode the optional operand Slit4 which determines the amount of the accumulator preshift; if the operand Slit4 is absent, a 0 is encoded.

See Table 1-7 for modifier addressing information.

**Note:** Positive values of operand Slit4 represent arithmetic shift right.  
Negative values of operand Slit4 represent shift left.

Words:	1
Cycles:	1

### Examples

Example1	LAC	A,W5	; Load Accumulator A
	Before Instruction		
	After Instruction		

09370457-060404

# LDW

Move f to Wn

Syntax: {label;} MOV f, Wn

Operands: f ∈ [0 ... 65535];  
Wn ∈ [W0 ... W15]

Operation: (f) → Wn

Status Affected: None

Encoding: 

1000	dddd	ffff	ffff	ffff	ffff
------	------	------	------	------	------

Description: Moves contents of any file register to a specified W register.

The 'f' bits select the address of the file register.  
The 'd' bits select the address of the destination register.

**Note:** This instruction only operates on word operands.

Words: 1  
Cycles: 1

## Examples

Example1                      MOV    RAM100,W6            ; Move RAM100 to W6  
Before Instruction  
  
After Instruction

**LDDW**

### Double word move from Ws to W register pair

Syntax:	{label:}	MOV.D	Ws,	Wrd
			[Ws],	
			[Ws]++,	
			[Ws]--,	
			[Ws++] ,	
			[Ws--] ,	
			none	

Operands:           Ws ∈ [W0 ... W15];  
                      Wnd ∈ [W0 ... W14]

**Operation:** See Section 5.6

Status Affected:           None

**Encoding:**

1011	1110	0000	0ddd	0ppp	ssss
------	------	------	------	------	------

<b>Description:</b>	This instruction supports fast context switch by loading a register pair in one cycle.
---------------------	--

The assembly mnemonic “POP.D Wnd” translates to the “LDDW (W15++),Wnd” instruction.

The 's' bits select the address of the first source register.

The 'd' bits select the address of the destination register. The least significant bit of the 'd' field must be '0'.

The 'p' bits select source address mode 2 (values 0-4).

See Table 1-5 for modifier addressing information.

**Note:** This instruction only operates on double word operands.

Words: 1

Cycles: 1

## Examples

Example1                      MOV.D    W6                      ; Pop W7 then W6 from stack

### Before Instruction

### After Instruction

# LDQW

Quad word move from Ws to W register quad

Syntax:	{label:}	MOV.Q	Ws,	Wnd
			[Ws],	
			[Ws]++,	
			[Ws]--,	
			[Ws++] ,	
			[Ws--] ,	

Operands: Ws ∈ [W0 ... W15];  
Wnd ∈ [W0,W4,W8,W12]

Operation: See Section 5.6

Status Affected: None

Encoding:	1011	1110	0100	0dd0	0ppp	ssss
-----------	------	------	------	------	------	------

Description: This instruction supports fast context switch by loading a register quad in two cycles.

The assembly mnemonic "POP.Q Wnd" translates to the "LDQW (W15++),Wnd" instruction.

The 's' bits select the address of the first source register.  
The 'd' bits select the address of the destination register. The least significant 2 bits of the 'd' field must be '0'.  
The 'p' bits select source address mode 2 (values 0-4).

See Table 1-5 for modifier addressing information.

**Note:** This instruction only operates on quad word operands.

Words: 1

Cycles: 1

## Examples

Example1 MOV.Q W4 ; Pop W7,W6,W5,W4 from stack

Before Instruction

After Instruction



LNK

Allocate Stack Frame

Syntax: {label;} LNK lit14

Operands: lit14 ∈ [0 ... 16384]

Operation: (W14) → [W15]--;  
(W15) → W14;  
(W15) - lit14 → W15

Status Affected: None

1111	1010	00kk	kkkk	kkkk	kkkk
------	------	------	------	------	------

Description: This instruction allocates a stack frame of size lit14 and adjusts the stack pointer and frame pointer.

The 'k' bits specify the size of the stack frame.

Words: 1

Cycles: 1

Examples

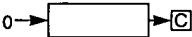
Example1

LSR

Logical Shift Right Ws

Syntax:	{label:}	LSR{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++] ,	[Wd++]
			[Ws--],	[Wd--]

Operands: Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]  
Operation: For word operation:  
0 → Wd<15>, (Ws<15:1>) → Wd<14:0>, (Ws<0>) → C  
For byte operation:  
0 → Wd<7>, (Ws<7:1>) → Wd<6:0>, (Ws<0>) → C



Status Affected: C, N, OV, Z

Encoding:	1101	0001	0Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Shift the contents of the source register Ws one bit to the right and place the result in the destination register Wd. The Carry Flag bit is set if the LSB of Ws is '1'.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select source address mode 2.
- The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

Examples

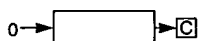
Example1 LSR W5,W6 ; Shift right  
Before Instruction  
  
After Instruction

### Logical Shift Right f

Syntax: {label:} LSR{.b} f {,Ww}

Operands:  $f \in [0 \dots 8191]$

Operation:            For word operation:  
                              0 → Dest<15>,    (f<15:1>) → Dest<14:0>,    (f<0>) → C  
                              For byte operation:  
                              0 → Dest<7>,    (f<7:1>) → Dest<6:0>,    (f<0>) → C



**Status Affected:** C, N, OV, Z

Encoding:	1101	0101	0BDf	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description:	Shift the contents of the file register <i>f</i> one bit to the right and place the result in the destination designated by <i>D</i> : If the optional <i>Ww</i> is specified, <i>D</i> =0 and store result in <i>Ww</i> ; otherwise, <i>D</i> =1 and store result in the file register. The carry flag bit is set if the LSB of the file register is '1'.
--------------	--

The 'B' bit selects byte or word operation.

The 'D' bit selects the destination.

The 's' bits select the address of the working register.

The 'f' bits select the address of the file register.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

**Example1**                  LSR      RAM135, Ww      ; Shift right

### Before Instruction

### After Instruction



09870457 "060101

# LSRW

Logical Shift Right by Wns

Syntax: {label:} LSR Wb, Wns, Wnd

Operands: Wb ∈ [W0 ... W15]; Wns ∈ [W0 ...W15]; Wnd ∈ [W0 ... W15]

Operation: Wns<3:0>→Shift\_Val

0→Shift\_In<39:32>  
Wb<15:0>→Shift\_In<31:16>  
0→Shift\_In<15:0>

0→Shift\_Out<39:32-Shift\_Val>  
Shift\_In<31:Shift\_Val>→Shift\_Out<31-Shift\_Val:0>

If Wns<4>==0: (less than 16)  
Shift\_Out<31:16>→Wnd  
Shift\_Out<15:0>→CARRY1  
0→CARRY0  
If Wns<4>==1: (16 or greater)  
0→Wnd<15:0>  
Shift\_Out<31:16>→CARRY1  
Shift\_Out<15:0>→CARRY0

Status Affected: C,SZ,Z

Encoding:	1101	1101	1www	wddd	d000	ssss
-----------	------	------	------	------	------	------

Description: Logical shift right the contents of the source register Wb by Wns bits (up to 31 positions), placing the result in the destination register Wnd. Bits that are shifted beyond the rightmost position of the source are stored in the CARRY1 and CARRY0 registers.

The Z and SZ bits will be set if the value placed in Wnd is zero and cleared otherwise. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.

**Note:** This instruction operates in word mode only.

Words: 1

Cycles: 1

## EXAMPLES:

# MAC

## Multiply and Accumulate

Syntax:	{label:} MAC	A, Wm*Wn	,Wxp,[Wx]	,Wyp,[Wy]	,AWB
		B,	,Wxp,[Wx]+=kx	,Wyp,[Wy]+=ky	<i>none</i>
			,Wxp,[Wx]-=kx ‡	,Wyp,[Wy]-=ky ‡	
			,Wxp,[W5+W8]	,Wyp,[W7+W8]	
			<i>none</i>	<i>none</i>	

‡ Alternate format for negative kx,ky

Operands:	$Wm * Wn \in \{W0 * W1; W0 * W2; W0 * W3; W1 * W2; W1 * W3; W2 * W3\}$ $Wxp \in \{W0 \dots W3\}; Wx \in \{W4, W5\}; kx \in \{-6, -4, -2, 2, 4, 6\};$ $Wyp \in \{W0 \dots W3\}; Wy \in \{W6, W7\}; ky \in \{-6, -4, -2, 2, 4, 6\};$ $AWB \in \{W9, [W9]++\}$						
Operation:	$(ACC(A \text{ or } B)) + (Wm) * (Wn) \rightarrow ACC(A \text{ or } B);$ $([Wx]) \rightarrow Wxp; (Wx) + kx \rightarrow Wx;$ $([Wy]) \rightarrow Wyp; (Wy) + ky \rightarrow Wy;$ $(ACC(B \text{ or } A)) \text{ rounded} \rightarrow AWB$						
Status Affected:	OA, OB, SA, SB						
Encoding:	<table><tr><td>1100</td><td>0mmn</td><td>A0xx</td><td>yyii</td><td>ijjj</td><td>jjaa</td></tr></table>	1100	0mmn	A0xx	yyii	ijjj	jjaa
1100	0mmn	A0xx	yyii	ijjj	jjaa		
Description:	<p>Signed, fractional or integer multiply the contents of two W registers. The 32-bit result is sign-extended to 40-bits and added to the specified accumulator.</p> <p>Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.</p> <p>Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required.</p> <p>AWB specifies the direct or indirect store of the convergently rounded contents of other accumulator, if required.</p> <p>The 'm' bits select the operand registers Wm and Wn for the multiply: The 'A' bit selects the accumulator for the result. The other accumulator is used for write back. The 'i' bits select the Wx pre-fetch operation. The 'j' bits select the Wy pre-fetch operation. The 'x' bits select the pre-fetch Wxp destination. The 'y' bits select the pre-fetch Wyp destination. The 'a' bits select the accumulator write-back destination.</p> <p>See Table 1-9 through Table 1-14 for modifier addressing information.</p>						
Words:	1						
Cycles:	1						

## 11

1

Before Instruction

ACCA = 2

ACCB = 3

$$W_0 = 5$$

**W1 = 6**

$$W2 = 7$$

**W3 = 8**

**W8 = 1000**

**W10 = 2000**

RAM(994) = 16

$$\text{RAM}(1000) = 17$$
$$\text{RAM}(2000) = 18$$

### After Instruction

$$ACCA = 2 + 7 \cdot 8 = 58$$

ACCB = 3

**W0 = 17**

**W1 = 18**

$$W_2 = 7$$

**W3 = 8**

**W8 = 994**

**W10 = 2000**

RAM(994) = 3

$$\text{RAM}(1000) = 17$$
$$\text{RAM}(2000) = 18$$

0590657-0601014

MOV

Move Ws to Wd

Syntax:	{label:}	MOV{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++	[Wd]++
			[Ws]--	[Wd]--
			[Ws--],	[Wd--]
			[Ws+Wb],	[Wd+Wb]
			[Ws+lit5],	[Wd+lit5]

Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]; Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]					
Operation:	(EAs) → EAd					
Status Affected:	None					
Encoding:	0111	1www	wBhh	hddd	dggg	ssss
Description:	Move the contents of the source register into the destination register.					

The 'B' bit selects byte or word operation.  
The 's' bits select the address of the source register.  
The 'd' bits select the address of the destination register.  
The 'g' bits select source address mode 3.  
The 'h' bits select destination address mode 3.  
The 'w' bits define the addressing mode literal 'lit5' or offset Wb; these bits are shared by source and destination addresses.

See Table 1-7 and Table 1-8 for modifier addressing information.

The assembly mnemonics PUSH Ws and POP Wd translate to MOV.

**Note:** The extension .b in the instruction denotes a byte move rather than a word move. You may use a .w extension to denote a word move, but it is not required.

Words:	1
Cycles:	1

Examples

Example1	MOV	W5,W6	; Move W5 to W6
Before Instruction			
After Instruction			



# NEGAB

## Negate Accumulators

Syntax: {label;} NEG A  
B

Operands: none

Operation: if (NEGAB A) then -ACCA → ACCA  
if (NEGAB B) then -ACCB → ACCB

Status Affected: OA, OB, SA, SB

Encoding: 

1100	1011	A001	0000	0000	0000
------	------	------	------	------	------

Description: Negate Accumulator.

The 'A' bits specify the selected accumulator.

Words: 1

Cycles: 1

### Examples

Example1                      NEG    B                      ; Negate ACCB, result to ACCB  
Before Instruction  
  
After Instruction

09870457-060104  
T.O.T.O.O "25402860

## Negate Ws

Syntax:	{label:}	NEG{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]

Operands:  $Ws \in [W0 \dots W15]; Wd \in [W0 \dots W15]$

Operation:  $\overline{(Ws)} + 1 \rightarrow Wd$

Status Affected: C, DC, N, OV, Z

Encoding:	1110	1010	0Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

**Description:** Compute the 2's complement of the contents of the source register Ws and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.

The 's' bits select the address of the source register.

The 'd' bits select the address of the destination register.

The 'p' bits select the source address mode 2 (values 0-4).

The 'q' bits select the destination address mode 2 (values 0-4).

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

Example1                      NEG            W5,W7                      ; Negate

### Before Instruction

### After Instruction

0970671

09870457 060404  
T0T090 2502860

# NEGF

Negate f

Syntax: {label:} NEG{.b} f {,Ww}

Operands:

f ∈ [0 ... 8191]

Operation:

$\overline{(f)} + 1 \rightarrow$  destination designated by D

Status Affected:

C, DC, N, OV, Z

Encoding:

1110	1110	0BDf	ffff	ffff	ffff
------	------	------	------	------	------

Description:

Compute the 2's complement of the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.  
The 'D' bit selects the destination.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:

1

Cycles:

1

## Examples

Example1

NEG

RAM135

; Negate

Before Instruction

After Instruction

# NOP

No Operation

Syntax: {label;} NOP

Operands:	None						
Operation:	No Operation						
Status Affected:	None						
Encoding:	<table border="1"><tr><td>0000</td><td>0000</td><td>xxxx</td><td>xxxx</td><td>xxxx</td><td>xxxx</td></tr></table>	0000	0000	xxxx	xxxx	xxxx	xxxx
0000	0000	xxxx	xxxx	xxxx	xxxx		
Description:	No Operation is performed.  The 'x' bits can take any value.						
Words:	1						
Cycles:	1						

## Examples

Example1	NOP	; No operation
	Before Instruction	
	After Instruction	

# NOPR

No Operation

Syntax: {label;} NOPR

Operands:	None						
Operation:	No Operation						
Status Affected:	None						
Encoding:	<table border="1"><tr><td>1111</td><td>1111</td><td>xxxx</td><td>xxxx</td><td>xxxx</td><td>xxxx</td></tr></table>	1111	1111	xxxx	xxxx	xxxx	xxxx
1111	1111	xxxx	xxxx	xxxx	xxxx		
Description:	No Operation is performed.  The 'x' bits can take any value.						
Words:	1						
Cycles:	1						

## Examples

Example1	NOPR	; No Opeation
	Before Instruction	
	After Instruction	

# POP

Pop top of Return Stack

Syntax: {label:} POP f

Operands: f ∈ [0 ... 65534]  
Operation: (W15)+2 → W15  
(TOS) → f

Status Affected: None

Encoding: 

1111	1001	ffff	ffff	ffff	ffff
------	------	------	------	------	------

Description: The stack pointer (W15) is pre-incremented and Top of Stack (TOS) value is pulled off the stack and written to the file register.

**Note:** This instruction operates in word mode only.

Words: 1  
Cycles: 1

## Examples

Example1                      POP      RAM135                      ; Pop  
Before Instruction  
  
After Instruction

### Push top of return stack (TOS)

Syntax:	{label:}	PUSH	f
---------	----------	------	---

Operands:  $f \in [0 \dots 65534]$

Operation: (f) → (TOS)  
(W15)-2 → W15

Status Affected:           None

**Encoding:**

1111	1000	ffff	ffff	ffff	ffff
------	------	------	------	------	------

**Description:**

The file register contents are written to the Top of Stack (TOS) location. Then the stack pointer (W15) is post decremented.

**Note:** This instruction operates in word mode only.

Words: 1

Cycles: 1

## Examples

**Example1**                      **PUSH    RAM135            ; Push**

Before Instruction

### After Instruction

# THE UNIVERSITY OF CHICAGO

## Relative Call

**Syntax:** {label:} RCALL Slit16

**Operands:** Slit16 ∈ [-32768 ... +32767]

Operation:

- (PC) + 2 → PC,
- (PC<15:0>) → TOS,
- (W15)+2 → W15
- (PC<23:16>) → TOS,
- (W15)+2 → W15
- (PC) + (2 \* Slit16) → PC, NOP → Instruction Register.

Status Affected:           None

Encoding:	0000	0111	nnnn	nnnn	nnnn	nnnn
-----------	------	------	------	------	------	------

Description:	Subroutine call with a jump up to 32K instructions from the current location. First, return address (PC+2) is pushed onto the return stack (20-bits wide).
--------------	--

Then the sign extended 17-bit value ( $2 * \text{Slit16}$ ) is added to the contents of the PC and the result is stored into the PC. RCALL is a two-cycle instruction.

Words: 1

Cycles: 2

## Examples

**Example1**                      RCALL    label                      ; Call subroutine

### Before Instruction

### After Instruction



09876543210 " 060101

# RCALLW

Computed Call

Syntax: {label:} RCALL Wn

Operands: Wn ∈ [W0 ... W15]  
Operation: (PC) +2 → PC,  
(PC<15:0>) → TOS,  
(W15)+2 → W15  
(PC<23:16>) → TOS,  
(W15)+2 → W15  
(PC) + (2 \* (Wn)) → PC, NOP → Instruction Register.

Status Affected: None

Encoding:	0000	0001	0010	0000	0000	ssss
-----------	------	------	------	------	------	------

Description: Computed subroutine call with a jump up to 32K instructions forward or back from the current location. First, return address (PC+2) is pushed onto the return stack.  
  
Then the sign extended 17-bit value (2 \* (Wn)) is added to the contents of the PC and the result is stored into the PC. RCALLW is a two-cycle instruction.

Words: 1  
Cycles: 2

## Examples

Example1                      RCALL    W11                      ; Call subroutine at PC+W11  
Before Instruction  
  
After Instruction

**Repeat next instruction n times**

**Syntax:** {label:} REPEAT lit14

Operands:  $\text{lit14} \in [1 \dots 16383]$

**Operation:** (lit14) → LCR (Loop Count Register)  
(PC)+2 → PC  
Enable Code Looping

Status Affected:           None

**Encoding:**

0000	1001	00kk	kkkk	kkkk	kkkk
------	------	------	------	------	------

**Description:**

The instruction immediately following the REPEAT instruction is repeated 16 times. The repeated instruction is held in the instruction register for all iterations and so is fetched only once (during the REPEAT instruction, as would be expected). The first iteration of the repeated instruction pre-fetches the next instruction.

The repeat count is decremented during each iteration. When it equals zero, the pre-fetch instruction is staged into the instruction and normal execution continues.

The repeated instruction can be interrupted before any iteration, but only by a priority 1 (fast context switch) interrupt. Subsequent interrupts must be held pending until the repeat operation is complete. Note that nested repeats (e.g. from within the interrupt service routine) are not supported.

The 'k' bits are an unsigned literal that specifies the loop count.

Words: 1

Cycles: 1 + lit14

## Examples

**Example1**                      REPEAT   #5                      ; Repeat next instruction 5 times

### Before Instruction

### After Instruction

**00000000000000000000000000000000**

### Move f to destination

**Syntax:**                    {label:}       MOV{.b}                f                {,Ww}

Operands:  $f \in [0 \dots 8191]$ ;

**Operation:** (f) → destination designated by D

Status Affected: Z, N

Encoding:

1011	1111	1BDf	ffff	ffff	ffff
------	------	------	------	------	------

**Description:**

Move the contents of the file register to the destination designated by D: if D=0, put the value into Ww, if D=1 the only effect is to modify the status flags, no writeback is required.

The 'B' bit selects byte or word operation.

The 'D' bit selects the destination, (0 for Wd, 1 for f).

The 'f' bits select the address of the file register.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

**Example1**                      **MOV**     **RAM433, Ww**        ; Move File register 433 to Ww

**Before Instruction**

### After Instruction

### Move 16-bit literal to Wd

Syntax:	{label:}	MOV	lit16,	Wn
---------	----------	-----	--------	----

**Operands:** lit16  $\in [-32768 \dots 65535]$ ; Wn  $\in [W0 \dots W15]$

Operation:  $\text{lit16} \rightarrow W_n$

Status Affected: None

**Encoding:**

0010	dddd	kkkk	kkkk	kkkk	kkkk
------	------	------	------	------	------

<b>Description:</b>	The Literal 'k' is loaded into Wn register.
---------------------	---

The 'd' bits select the address of the working register.  
The 'k' bits specify the value of the literal.

Words: 1

Cycles: 1

## Examples

**Example1**                      MOV    #64159, W5            ; Move 64159 into W5

### Before Instruction

### After Instruction

### Move literal to Wn

**Syntax:** {label:} MOV{.b} Slit10, Wn

Operands: Slit10  $\in [-512 \dots 511]$ ; Wn  $\in [W0 \dots W15]$

Operation: Slit10  $\rightarrow$  Wn

Status Affected: None

**Encoding:**

1011	0011	1Bdd	ddkk	kkkk	kkkk
------	------	------	------	------	------

Description:	The Literal 'k' is loaded into Wn register.
--------------	---

The 'B' bit selects byte or word operation.

The 'd' bits select the address of the working register.

The 'k' bits specify the value of the literal.

**Note:** The extension `.b` in the instruction denotes a byte move rather than a word move. You may use a `.w` extension to denote a word move, but it is not required.

**Words:** 1

Cycles: 1

## Examples

**Example1**                      MOV    #159, W5                      ; Move 159 into W5

### Before Instruction

### After Instruction

MOVSAC

Prefetch Operands and Store Accumulator

Syntax:	{label:} MOVSAC	A,	,Wxp,[Wx]	,Wyp,[Wy]	,AWB
		B,	,Wxp,[Wx]+=kx	,Wyp,[Wy]+=ky	none
			,Wxp,[Wx]-=kx ‡	,Wyp,[Wy]-=ky ‡	
			,Wxp,[W5+W8]	,Wyp,[W7+W8]	
			none	none	

‡ Alternate format for negative kx,ky

Operands: Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6};  
Wyp ∈ {W0 ... W3}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6};  
AWB ∈ {W9, [W9]++}

Operation: ([Wx])→ Wxp; (Wx)+kx→Wx;  
([Wy])→ Wyp; (Wy)+ky→Wy;  
(ACC(B or A)) rounded → AWB

Status Affected: OA, OB, SA, SB

Encoding:	1100	0111	A0xx	yyii	ijjj	jjaa
-----------	------	------	------	------	------	------

Description: Prefetch operands and optionally store accumulator results in preparation for a repeated MAC type instruction.  
Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.  
Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required.  
AWB specifies the direct or indirect store of the convergently rounded contents of other accumulator, if required. Note that the specification of (B or A) is consistent with the MAC instruction. For example, MOVSAC A, W9 will store ACCB into W9.

- The 'A' bit selects the other accumulator used for write back.
- The 'i' bits select the Wx pre-fetch operation.
- The 'j' bits select the Wy pre-fetch operation.
- The 'x' bits select the pre-fetch Wxp destination.
- The 'y' bits select the pre-fetch Wyp destination.
- The 'a' bits select the accumulator write-back destination.

See Table 1-9 through Table 1-14 for modifier addressing information.

Words: 1  
Cycles: 1

## Examples

Example1      MOV SAC    A,W0,[W4]-=6,W1,[W6],W9      ; Prefetch and move ACCB to W9

Before Instruction

ACCA = 2

ACCB = 3

W0 = 5

W1 = 6

W2 = 7

W3 = 8

W8 = 1000

W10 = 2000

RAM(994) = 16

RAM(1000) = 17

RAM(2000) = 18

After Instruction

09870457-060404

# MOVWF

Move Ww to F

Syntax: {label:} MOV{.b} Ww, f

Operands: f ∈ [0 ... 8191]

Operation: (Ww) → f

Status Affected: None

Encoding:	1011	0111	1B1f	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description: Move the contents of the working register into the file register.

The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte move rather than a word move. You may use a .w extension to denote a word move, but it is not required.

Words: 1

Cycles: 1

## Examples

Example1 MOV Ww,213 ; Move Ww to File Register 213

Before Instruction

After Instruction



109876543210

MPY

Multiply Wm by Wn to Accumulator

Syntax:	{label:} MPY	A, Wm*Wn	,Wxp,[Wx]	,Wyp,[Wy]
		B,	,Wxp,[Wx]+=kx	,Wyp,[Wy]+=ky
			,Wxp,[Wx]-=kx ‡	,Wyp,[Wy]-=ky ‡
			,Wxp,[W5+W8]	,Wyp,[W7+W8]
			none	none

‡ Alternate format for negative kx,ky

Operands: Wm\*Wn ∈ {W0\*W1; W0\*W2; W0\*W3; W1\*W2; W1\*W3; W2\*W3}  
Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6};  
Wyp ∈ {W0 ... W3}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6};  
AWB ∈ {W9, [W9]++}

Operation: (Wm)\*(Wn) → ACC(A or B);  
([Wx])→ Wxp; (Wx)+kx→Wx;  
([Wy])→ Wyp; (Wy)+ky→Wy;

Status Affected: OA, OB, SA, SB

Encoding:	1100	0mmm	A0xx	yyii	ijjj	jj11
-----------	------	------	------	------	------	------

Description: Signed, fractional or integer multiply the contents of two W registers. The 32-bit result is sign-extended to 40-bits and stored to the specified accumulator.  
Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.  
Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required.

The 'm' bits select the operand registers Wm and Wn for the multiply:  
The 'A' bit selects the accumulator for the result.  
The 'i' bits select the Wx pre-fetch operation.  
The 'j' bits select the Wy pre-fetch operation.  
The 'x' bits select the pre-fetch Wxp destination.  
The 'y' bits select the pre-fetch Wyp destination.

See Table 1-9 through Table 1-13 for modifier addressing information.

Words: 1  
Cycles: 1

030620Z FEB 68

**Example1**    MPY    A,W2\*W3,W0,[W5]=-6,W1,[W7]    ; Multiply into Accumulator A

Before Instruction

ACCA = 2

ACCB = 3

$W_0 = 5$

**W1 = 6**

**W2 = 7**

$$W_3 = 8$$

**W8 = 1000**

**W10 = 2000**

RAM(994) = 16

$$\text{RAM}(1000) = 17$$
$$\text{RAM}(2000) = 18$$

### After Instruction

ACCA = 7\*8=56

ACCB = 3

**W0 = 17**

**W1 = 18**

$$W_2 = 7$$
$$W_3 = 8$$

**W8 = 994**

**W10 = 2000**

RAM(994) = 16

$$\text{RAM}(1000) = 17$$
$$\text{RAM}(2000) = 18$$

### Multiply -Wm by Wn to Accumulator

Syntax:	{label:} MPYN	A, Wm*Wn	,Wxp,[Wx]	,Wyp,[Wy]
		B,	,Wxp,[Wx]+=kx	,Wyp,[Wy]+=ky
			,Wxp,[Wx]-=kx ‡	,Wyp,[Wy]-=ky ‡
			,Wxp,[W5+W8]	,Wyp,[W7+W8]
			<i>none</i>	<i>none</i>

‡ Alternate format for negative  $k_x, k_y$

Operands:  $Wm * Wn \in \{W0 * W1; W0 * W2; W0 * W3; W1 * W2; W1 * W3; W2 * W3\}$   
 $Wxp \in \{W0 \dots W3\}; Wx \in \{W4, W5\}; kx \in \{-6, -4, -2, 2, 4, 6\};$   
 $Wyp \in \{W0 \dots W3\}; Wy \in \{W6, W7\}; ky \in \{-6, -4, -2, 2, 4, 6\};$   
 $AWB \in \{W9, [W9]_{++}\}$

Operation:  $\neg(Wm) \cdot (Wn) \rightarrow ACC(A \text{ or } B);$   
 $([Wx]) \rightarrow Wxp; (Wx) + kx \rightarrow Wx;$   
 $([Wy]) \rightarrow Wyp; (Wy) + ky \rightarrow Wy;$

**Status Affected:** OA, OB, SA, SB

Encoding:	1100	0mmn	A1xx	yyii	iijj	jj11
-----------	------	------	------	------	------	------

Description:	<p>Signed, fractional or integer multiply the contents of a W register by the negative of the contents of another W register. The 32-bit result is sign-extended to 40-bits and stored to the specified accumulator.</p> <p>Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.</p> <p>Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required.</p>
--------------	---

The 'm' bits select the operand registers Wm and Wn for the multiply:

The 'A' bit selects the accumulator for the result.

The 'i' bits select the Wx pre-fetch operation.

The 'j' bits select the Wy pre-fetch operation.

The 'x' bits select the pre-fetch Wxp destination.

The 'y' bits select the pre-fetch Wyp destination.

See Table 1-9 through Table 1-13 for modifier addressing information.

**Words:** 1

Cycles: 1

09870457 "060101

MSC

Multiply and Subtract from Accumulator

Syntax:	{label:} MSC	A, Wm*Wn	,Wxp,[Wx]	,Wyp,[Wy]	,AWB
		B,	,Wxp,[Wx]+=kx	,Wyp,[Wy]+=ky	none
			,Wxp,[Wx]-=kx ‡	,Wyp,[Wy]-=ky ‡	
			,Wxp,[W5+W8]	,Wyp,[W7+W8]	
			none	none	

‡ Alternate format for negative kx,ky

Operands: Wm\*Wn ∈ {W0\*W1; W0\*W2; W0\*W3; W1\*W2; W1\*W3; W2\*W3}  
Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6};  
Wyp ∈ {W0 ... W3}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6};  
AWB ∈ {W9, [W9]++}

Operation: (ACC(A or B)) – (Wm)\*(Wn) → ACC(A or B);  
([Wx])→ Wxp; (Wx)+kx→Wx;  
([Wy])→ Wyp; (Wy)+ky→Wy;  
(ACC(B or A)) rounded → AWB

Status Affected: OA, OB, SA, SB

Encoding:	1100	0mmn	A1xx	yyii	iijj	jjaa
-----------	------	------	------	------	------	------

Description: Signed, fractional or integer multiply the contents of two W registers. The 32-bit result is sign-extended to 40-bits and subtracted from the specified accumulator.  
Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.  
Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required.  
AWB specifies the direct or indirect store of the convergently rounded contents of other accumulator, if required.

The 'm' bits select the operand registers Wm and Wn for the multiply:  
The 'A' bit selects the accumulator for the result. The other accumulator is used for write back.  
The 'i' bits select the Wx pre-fetch operation.  
The 'j' bits select the Wy pre-fetch operation.  
The 'x' bits select the pre-fetch Wxp destination.  
The 'y' bits select the pre-fetch Wyp destination.  
The 'a' bits select the accumulator write-back destination.

See Table 1-9 through Table 1-14 for modifier addressing information.

Words: 1  
Cycles: 1

## Examples

Example1    MPYN    A,W2\*W3,W0,[W4]-=6,W1,[W6]    ; Multiply negative into Acc A

Before Instruction

ACCA = 2

ACCB = 3

W0 = 5

W1 = 6

W2 = 7

W3 = 8

W8 = 1000

W10 = 2000

RAM(994) = 16

RAM(1000) = 17

RAM(2000) = 18

After Instruction

ACCA = -7\*8=-56

ACCB = 3

W0 = 17

W1 = 18

W2 = 7

W3 = 8

W8 = 994

W10 = 2000

RAM(994) = 16

RAM(1000) = 17

RAM(2000) = 18

## Examples

Example1    MSC    A, W2\*W3, W0=[W4]-=6, W1=[W6], W9    ; Multiply and Subtract A

Before Instruction

ACCA = 2

ACCB = 3

W0 = 5

W1 = 6

W2 = 7

W3 = 8

W8 = 1000

W10 = 2000

RAM(994) = 16

RAM(1000) = 17

RAM(2000) = 18

After Instruction

ACCA =  $2 + 7 * 8 = 58$

ACCB = 3

W0 = 17

W1 = 18

W2 = 7

W3 = 8

W8 = 994

W10 = 2000

RAM(994) = 3

RAM(1000) = 17

RAM(2000) = 18

09370457-060101  
"25402860"

### Multi-Byte Shift Left by Short Literal

Syntax: {label:} MSL Wb, lit5, Wnd

**Operands:** Wb  $\in$  [W0 ... W15]; k  $\in$  [0...31]; Wnd  $\in$  [W0 ... W15]

Operation:  $\text{lit5} \langle 3:0 \rangle \rightarrow \text{Shift\_Val}$

0→Shift\_In<39:16>

**Wb<15:0>→Shift\_In<15:0>**

0→Shift\_Out<39:16+Shift\_Val>

Shift\_In<15:0l>→Shift\_Out<15+Shift\_Val:Shift\_Val>

If lit5<4>==0: (less than 16)

0→CARRY1<15:0>

Shift\_Out<31:16> .OR. CARRY1<15:0>→CARRY0<15:0>

Shift\_Out<15:0> .OR. CARRY0<15:0>→Wnd<15:0>

If lit5<4>==1: (16 or greater)

Shift\_Out<31:16>→CARRY1<15:0>

Shift\_Out<15:0> .OR. CARRY1<15:0>→CARRY0<15:0>

0 .OR. CARRY0<15:0>→Wnd<15:0>

Status Affected: C,SZ,Z

Encoding:

1101	1100	0www	wddd	d11k	kkkk
------	------	------	------	------	------

**Description:**

Shift left the contents of the source register Wb by lit5 bits (up to 31 positions), OR in the contents of the CARRY1 and CARRY0 registers then place the result in the destination register Wnd. Bits that are shifted beyond the leftmost position of the source are stored in the CARRY1 and CARRY0 registers.

The Z bit will be set if the value placed in Wnd is zero and cleared otherwise. The SZ bit will be cleared if the value placed in Wnd is not zero. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.

**Note:** This instruction operates in word mode only.

Words: 1

Cycles: 1

**EXAMPLES:**

707090"25402860

# MSLW

## Multi-Byte Shift Left by Wns

Syntax: {label:} MSL Wb, Wns, Wnd

Operands: Wb ∈ [W0 ... W15]; Wns ∈ [W0 ...W15]; Wnd ∈ [W0 ... W15]

Operation: Wns<3:0>→Shift\_Val

0→Shift\_In<39:16>

Wb<15:0>→Shift\_In<15:0>

0→Shift\_Out<39:16+Shift\_Val>

Shift\_In<15:0>→Shift\_Out<15+Shift\_Val:Shift\_Val>

If Wns<4>==0: (less than 16)

0→CARRY1<15:0>

Shift\_Out<31:16> .OR. CARRY1<15:0>→CARRY0<15:0>

Shift\_Out<15:0> .OR. CARRY0<15:0>→Wnd<15:0>

If Wns<4>==1: (16 or greater)

Shift\_Out<31:16>→CARRY1<15:0>

Shift\_Out<15:0> .OR. CARRY1<15:0>→CARRY0<15:0>

0 .OR. CARRY0<15:0>→Wnd<15:0>

Status Affected:

C,SZ,Z

Encoding:

1101	1100	0www	wddd	d000	ssss
------	------	------	------	------	------

Description:

Shift left the contents of the source register Wb by Wns bits (up to 31 positions), OR in the contents of the CARRY1 and CARRY0 registers then place the result in the destination register Wnd. Bits that are shifted beyond the leftmost position of the source are stored in the CARRY1 and CARRY0 registers.

The Z bit will be set if the value placed in Wnd is zero and cleared otherwise. The SZ bit will be cleared if the value placed in Wnd is not zero. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.

**Note:** This instruction operates in word mode only.

Words: 1

Cycles: 1

### EXAMPLES:



09370457 060101

# MSRK

## Multi-Byte Shift Right by Short Literal

Syntax: {label:} MSR Wb, lit5, Wnd

Operands: Wb ∈ [W0 ... W15]; lit5 ∈ [0...31]; Wnd ∈ [W0 ... W15]  
Operation: lit5<3:0>→Shift\_Val

0→Shift\_In<39:32>  
Wb<15:0>→Shift\_In<31:16>  
0→Shift\_In<15:0>  
  
0→Shift\_Out<39:32-Shift\_Val>  
Shift\_In<31:Shift\_Val>→Shift\_Out<31-Shift\_Val:0>  
  
If lit5<4>==0: (less than 16)  
    Shift\_Out<31:16> .OR. CARRY1<15:0>→Wnd<15:0>  
    Shift\_Out<15:0> .OR. CARRY0<15:0>→CARRY1<15:0>  
    0→CARRY0<15:0>  
If lit5<4>==1: (16 or greater)  
    CARRY1<15:0>→Wnd<15:0>  
    Shift\_Out<31:16> .OR. CARRY0<15:0>→CARRY1<15:0>  
    Shift\_Out<15:0>→CARRY0<15:0>

Status Affected:

C,SZ,Z

Encoding:

1101	1100	1www	wddd	d11k	kkkk
------	------	------	------	------	------

Description:

Shift right the contents of the source register Wb by lit5 bits (up to 31 positions), OR in the contents of the CARRY1 and CARRY0 registers then place the result in the destination register Wnd. Bits that are shifted beyond the rightmost position of the source are stored in the CARRY1 and CARRY0 registers.  
  
The Z bit will be set if the value placed in Wnd is zero and cleared otherwise. The SZ bit will be cleared if the value placed in Wnd is not zero. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.

**Note:** This instruction operates in word mode only.

Words: 1  
Cycles: 1

### EXAMPLES:

09670457 "060101

# MSRW

## Multi-Byte Shift Right by Wns

Syntax: {label:} MSR Wb, Wns, Wnd

Operands: Wb ∈ [W0 ... W15]; Wns ∈ [W0 ...W15]; Wnd ∈ [W0 ... W15]  
Operation: Wns<3:0>→Shift\_Val

0→Shift\_In<39:32>  
Wb<15:0>→Shift\_In<31:16>  
0→Shift\_In<15:0>  
  
0→Shift\_Out<39:32-Shift\_Val>  
Shift\_In<31:Shift\_Val>→Shift\_Out<31-Shift\_Val:0>  
  
If Wns<4>==0: (less than 16)  
    Shift\_Out<31:16> .OR. CARRY1<15:0>→Wnd<15:0>  
    Shift\_Out<15:0> .OR. CARRY0<15:0>→CARRY1<15:0>  
    0→CARRY0<15:0>  
If Wns<4>==1: (16 or greater)  
    CARRY1<15:0>→Wnd<15:0>  
    Shift\_Out<31:16> .OR. CARRY0<15:0>→CARRY1<15:0>  
    Shift\_Out<15:0>→CARRY0<15:0>

Status Affected: C,SZ,Z

Encoding:	1101	1100	1www	wddd	d000	ssss
-----------	------	------	------	------	------	------

Description: Shift right the contents of the source register Wb by Wns bits (up to 31 positions), OR in the contents of the CARRY1 and CARRY0 registers then place the result in the destination register Wnd. Bits that are shifted beyond the rightmost position of the source are stored in the CARRY1 and CARRY0 registers.

The Z bit will be set if the value placed in Wnd is zero and cleared otherwise. The SZ bit will be cleared if the value placed in Wnd is not zero. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.

**Note:** This instruction operates in word mode only.

Words: 1  
Cycles: 1

### EXAMPLES:

### 16x16 bit Signed Multiply

Syntax:	{label:}	MUL.SS	Wb,	Ws,	Wnd
				[Ws],	
				[Ws]++,	
				[Ws]--,	
				[Ws++] ,	
				[Ws--],	

Operands:           Wb ∈ [W0 ... W15];  
                          Ws ∈ [W0 ... W15];  
                          Wnd ∈ [W0,W2,W4,W6,W8,W10,W12,W14]

Operation: signed (Wb) \* signed (Ws)  $\rightarrow$  {Wnd+1, Wnd}

Status Affected:           None

Encoding:	1011	1001	1www	wddd	dppp	ssss
-----------	------	------	------	------	------	------

Description:	MULS performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.
--------------	---

Both source operands are interpreted as two's-complement signed integers.

The 'w' bits select the address of the base register

The 's' bits select the address of the source register.

The 'p' bits select source address mode 2.

The 'd' bits select the address of the destination for the product LSBs, the register 'd+1' is the destination of the product MSBs.

See Table 1-5 for modifier addressing information.

**Note:** This instruction operates in word mode only.

Words: 1

Cycles: 1

## Examples

Example1                      MUL.SS   W5, W6, W8                      ; Multiply W5\*W6 to W9:W8

### Before Instruction

### After Instruction

### 16x16 bit Signed-Unsigned Multiply

Syntax:	{label:}	MUL.SU	Wb,	Ws,	Wnd
				[Ws],	
				[Ws]++,	
				[Ws]--,	
				[Ws++] ,	
				[Ws--],	

Operands:	Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wnd ∈ [W0,W2,W4,W6,W8,W10,W12,W14]
Operation:	signed (Wb) * unsigned (Ws) → {Wnd+1, Wnd}
Status Affected:	None

Encoding:	1011	1001	0www	wddd	dppp	ssss
-----------	------	------	------	------	------	------

Description:	MULSU performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.
--------------	--

The first source operand is interpreted as a two's-complement signed integer and the second source operand is interpreted as an unsigned integer.

The 'w' bits select the address of the base register

The 's' bits select the address of the source register.

The 'p' bits select source address mode 2.

The 'd' bits select the address of the destination for the product LSBs, the register 'd+1' is the destination of the product MSBs.

See Table 1-5 for modifier addressing information.

**Note:** This instruction operates in word mode only.

Words: 1  
Cycles: 1

## Examples

**Example1**                      MUL.SU   W5, W6, W8                      ; Multiply W5\*W6 to W9:W8

### Before Instruction

### After Instruction

# MULSULS

16x16 bit Signed Multiply Unsigned Short Literal

Syntax: {label:} MUL.SU Wb, lit5, Wnd

Operands: Wb ∈ [W0 ... W15];  
lit5 ∈ [0 ... 31];  
Wnd ∈ [W0,W2,W4,W6,W8,W10,W12,W14]

Operation: signed (Wb) \* unsigned lit5 → {Wnd+1, Wnd}

Status Affected: None

Encoding:

1011	1001	1www	wddd	d11k	kkkk
------	------	------	------	------	------

Description: MULSULS performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.

The source operands is interpreted as a two's-complement signed integer and the literal is interpreted as an unsigned integer.

The 'k' bits define a 5-bit unsigned integer literal.  
The 'w' bits select the address of the base register.  
The 'd' bits select the address of the destination for the product LSBs, the register 'd+1' is the destination of the product MSBs.

**Note:** This instruction operates in word mode only.

Words: 1  
Cycles: 1

## Examples

Example1 MUL.SU W6, #13, W8 ; Multiply W6 times 13 into W9:W8

Before Instruction

After Instruction

# MULU

16x16 bit Unsigned Multiply

Syntax:	{label:}	MUL.UU	Wb,	Ws,	Wnd
				[Ws],	
				[Ws]++,	
				[Ws]--,	
				[Ws++] ,	
				[Ws--],	

Operands:	Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wnd ∈ [W0,W2,W4,W6,W8,W10,W12,W14]						
Operation:	unsigned (Wb) * unsigned (Ws) → {Wnd+1, Wnd}						
Status Affected:	None						
Encoding:	<table><tr><td>1011</td><td>1000</td><td>0www</td><td>wddd</td><td>dppp</td><td>ssss</td></tr></table>	1011	1000	0www	wddd	dppp	ssss
1011	1000	0www	wddd	dppp	ssss		
Description:	MULU performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.						

Both source operands are interpreted as unsigned integers.

The 'w' bits select the address of the base register.

The 's' bits select the address of the source register.

The 'p' bits select source address mode 2.

The 'd' bits select the address of the destination for the product LSBs, the register 'd+1' is the destination of the product MSBs.

See Table 1-5 for modifier addressing information.

**Note:** This instruction operates in word mode only.

Words:	1
Cycles:	1

## Examples

Example1	MUL.UU W5, W6, W8 ; Multiply W5*W6 to W9:W8
	Before Instruction
	After Instruction

# MULULS

16x16 bit Unsigned Multiply Short Literal

Syntax: {label:} MULULS Wb, lit5, Wnd

Operands: Wb ∈ [W0 ... W15];  
 lit5 ∈ [0 ... 31];  
 Wnd ∈ [W0,W2,W4,W6,W8,W10,W12,W14]

Operation: unsigned (Wb) \* unsigned lit5 → {Wnd+1, Wnd}

Status Affected: None

Encoding:

1011	1000	0www	wddd	d11k	kkkk
------	------	------	------	------	------

Description: MULULS performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.

Both operands are interpreted as unsigned integers.

The 'k' bits define a 5-bit unsigned integer literal..

The 'w' bits select the address of the base register.

The 'd' bits select the address of the destination for the product LSBs, the register 'd+1' is the destination of the product MSBs.

**Note:** This instruction operates in word mode only.

Words: 1  
 Cycles: 1

## Examples

Example1 MUL.UU W6, #13, W8 ; Multiply W6 times 13 into W9:W8

Before Instruction

After Instruction

MULUS

16x16 bit Unsigned-Signed Multiply

Syntax:	{label:}	MUL.US	Wb,	Ws,	Wnd
				[Ws],	
				[Ws]++,	
				[Ws]--,	
				[Ws++] ,	
				[Ws--],	

Operands:	Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wnd ∈ [W0,W2,W4,W6,W8,W10,W12,W14]						
Operation:	unsigned (Wb) * signed (Ws) → {Wnd+1, Wnd}						
Status Affected:	None						
Encoding:	<table><tr><td>1011</td><td>1000</td><td>1www</td><td>wddd</td><td>dppp</td><td>ssss</td></tr></table>	1011	1000	1www	wddd	dppp	ssss
1011	1000	1www	wddd	dppp	ssss		
Description:	MULUS performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.						

The first source operands is interpreted as an unsigned integer and the second source operand is interpreted as a two's-complement signed integer.

The 'w' bits select the address of the base register.  
The 's' bits select the address of the source register.  
The 'p' bits select source address mode 2.  
The 'd' bits select the address of the destination for the product LSBs, the register 'd+1' is the destination of the product MSBs.

See Table 1-5 for modifier addressing information.

**Note:** This instruction operates in word mode only.

Words:	1
Cycles:	1

Examples

Example1	MUL.US W5, W6, W8 ; Multiply W5*W6 to W9:W8
	Before Instruction
	After Instruction



# MULWF

8-bit x 8-bit Multiply

Syntax: {label:} MUL{.b} f

Operands:	f ∈ [0 ... 8191]					
Operation:	If byte mode, (Ww)<7:0> * (f)<7:0> → W2 If word mode, (Ww) * (f) → W3:W2					
Status Affected:	None					
Encoding:	1011	1100	0B0f	ffff	ffff	ffff
Description:	Multiply the working register and the file register and place the result in the W3:W2 register pair.  The 'B' bit selects byte or word operation. The 'f' bits select the address of the file register.  <b>Note:</b> Word operation is assumed.					
Words:	1					
Cycles:	1					

## Examples

Example1	MUL	RAM135	; Multiply Ww by RAM135
	Before Instruction		
	After Instruction		

**09876543210**

Syntax:                    {label:}       REPEAT               Wn

### After Instruction

# RESET

## Reset

**Syntax:** {label:} RESET

Operands: none

**Operation:** Force all registers and flag bits that are affected by a  $\overline{\text{MCLR}}$  reset to their reset condition.

Status Affected:           None

**Encoding:**

1111	1110	0000	0000	0000	0000
------	------	------	------	------	------

**Description:** This instruction provides a way to execute a software reset.

Words: 1

Cycles: 1

## Examples

```
Example1          RESET          ; Reset
```

### Before Instruction

### After Instruction

[illegible]

# RET FIE

Return from Interrupt

Syntax: {label:} RET FIE  
RET FIE.S

Operands: None  
Operation: (W15)-2 → W15  
TOS → (PC<23:16>),  
(W15)-2 → W15  
TOS → (PC<15:0>),  
NOP → Instruction Register.  
<Interrupt Flag Stuff - TBD>  
If S = 1,  
copy the contents of the shadow registers into the primary registers.

Status Affected: INTLV  
Encoding: 0000 0110 S100 0000 0000 0000  
Description: Return from interrupt service routine. The stack is popped and the Top of Stack (TOS) is loaded into the program counter. If 'S' = 1, the contents of the shadow registers are copied into the respective primary registers. If 'S' = 0, no update of these registers occurs (default). The Interrupt Level Register is updated.

Words: 1  
Cycles: 2

## Examples

Example1 RET FIE ; Return from interrupt  
Before Instruction  
After Instruction

# RETLW

Return with Literal in Wd

Syntax: (label:) RETLW{.b} Slit10, Wn  
RETLW.S

Operands: Wn ∈ [W0 ... W15]; Slit10 ∈ [-512 ... 511]

Operation: (W15)-2 → W15  
TOS → (PC<23:16>),  
(W15)-2 → W15  
TOS → (PC<15:0>),  
Slit10 → Wn  
If S = 1,  
copy the contents of the shadow registers into the primary registers.

Status Affected: None

Encoding: 

0000	0101	SBkk	kkkk	kkkk	dddd
------	------	------	------	------	------

Description: Return with a literal value in Wn.

The 'B' bit selects byte or word operation.  
The 'S' bit shadow pop.  
The 'd' bits select the address of the destination register.  
The 'k' bits define the literal.

Words: 1

Cycles: 2

## Examples

Example1 RETLW #-13, W5 ; Return  
Before Instruction

After Instruction

# RETURN

## Return

Syntax:

{label:}
RETURN
RETURN.S

Operands:

None

Operation:

(W15)-2 → W15  
TOS → (PC<23:16>),  
(W15)-2 → W15  
TOS → (PC<15:0>),  
NOP → Instruction Register.  
If S = 1,  
copy the contents of the shadow registers into the primary registers.

Status Affected:

None

Encoding:

0000	0110	S000	0000	0000	0000
------	------	------	------	------	------

Description:

Return from subroutine. The stack is popped and the Top of Stack (TOS) is loaded into the program counter. If 'S' = 1, the contents of the shadow registers are copied into the respective primary registers. If 'S' = 0, no update of these registers occurs (default).

Words:

1

Cycles:

2

### Examples

Example1

RETURN

; Return

Before Instruction

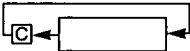
After Instruction

# RLC

Rotate Left Ws through Carry

Syntax:	{label:}	RLC{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]

Operands: Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]  
Operation: For word operation:  
(C) → Wd<0>, (Ws<14:0>) → Wd<15:1>, (Ws<15>) → C  
For byte operation:  
(C) → Wd<0>, (Ws<6:0>) → Wd<7:1>, (Ws<7>) → C



Status Affected:	C, N, Z					
Encoding:	1101	0010	1Bqq	qddd	dppp	ssss
Description:	Rotate the contents of the source register Ws one bit to the left through the carry flag and place the result in the destination register Wd.					

The 'B' bit selects byte or word operation.  
The 's' bits select the address of the source register.  
The 'd' bits select the address of the destination register.  
The 'p' bits select source address mode 2.  
The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1                      RLC      W5,W6                      ; Rotate left

Before Instruction

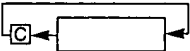
After Instruction

# RLCF

Rotate Left f through Carry

Syntax: {label:} RLC{.b} f {,Ww}

Operands: f ∈ [0 ... 8191]  
Operation: For word operation:  
(C) → Dest<0>, (f<14:0>) → Dest<15:1>, (f<15>) → C  
For byte operation:  
(C) → Dest<0>, (f<6:0>) → Dest<7:1>, (f<7>) → C



Status Affected: C, N, Z  
Encoding: 

1101	0110	1BDf	ffff	ffff	ffff
------	------	------	------	------	------

  
Description: Rotate the contents of the file register f one bit to the left through the carry flag and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination.  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

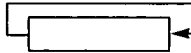
Example1 RLC RAM135, Ww ; Rotate left  
Before Instruction  
  
After Instruction



**098264**

Syntax:	{label:}	RLNC{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++] ,	[Wd++]
			[Ws--] ,	[Wd--]

Operation:            For word operation:  
                               (Ws<14:0>) → Wd<15:1>, (Ws<15>) → Wd<0>  
                               For byte operation:  
                               (Ws<6:0>) → Wd<7:1>,      (Ws<7>) → Wd<0>



Encoding:	1101	0010	0Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select source address mode 2.
- The 'q' bits select destination address mode 2.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

## Examples

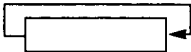
### After Instruction

# RLNCF

Rotate Left f (No Carry)

Syntax: {label:} RLNC{.b} f {,Ww}

Operands: f ∈ [0 ... 8191]  
Operation: For word operation:  
(f<14:0>) → Dest<15:1>, (f<15>) → Dest<0>  
For byte operation:  
(f<6:0>) → Dest<7:1>, (f<7>) → Dest<0>



Status Affected: N, Z

1101	0110	0BDf	ffff	ffff	ffff
------	------	------	------	------	------

Description: Rotate the contents of the file register f one bit to the left and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register. The carry flag bit is not affected.

The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination.  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

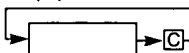
## Examples

Example1 RLNC RAM135, Ww ; Rotate left  
Before Instruction  
  
After Instruction

# RRC

## Rotate Right Ws through Carry

Syntax:	{label:}	RRC{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]

Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	For word operation: (C) → Wd<15>, (Ws<15:1>) → Wd<14:0>, (Ws<0>) → C For byte operation: (C) → Wd<7>, (Ws<7:1>) → Wd<6:0>, (Ws<0>) → C 					
Status Affected:	C, N, Z					
Encoding:	1101	0011	1Bqq	qddd	dppp	ssss
Description:	Rotate the contents of the source register Ws one bit to the right through the carry flag and place the result in the destination register Wd.					

Words:	1
Cycles:	1

### Examples

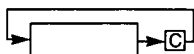
Example1	RRC	W5,W6	; Rotate right
	Before Instruction		
	After Instruction		

### Rotate Right f through Carry

Syntax: {label:} RRC{.b} f {,Ww}

Operands:  $f \in [0 \dots 8191]$

Operation: For word operation:  
(C) → Dest<15>, (f<15:1>) → Dest<14:0>, (f<0>) → C  
For byte operation:  
(C) → Dest<7>, (f<7:1>) → Dest<6:0>, (f<0>) → C



**Status Affected:** C, N, Z

**Encoding:**

1101	0111	1BDf	ffff	ffff	ffff
------	------	------	------	------	------

**Description:**

Rotate the contents of the file register f one bit to the left through the carry flag and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register..

The 'B' bit selects byte or word operation.

The 'D' bit selects the destination.

The 'f' bits select the address of the file register.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

**Example1**                      RRC      RAM135, Ww      ; Rotate right

### Before Instruction

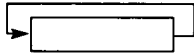
### After Instruction

### Rotate Right Ws (No Carry)

Syntax:	{label:}	RRNC{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			{Ws}--,	{Wd}--
			[Ws++] ,	[Wd++]
			[Ws--],	[Wd--]

Operands:  $Ws \in [W0 \dots W15]; Wd \in [W0 \dots W15]$

Operation:            For word operation:  
                               (Ws<15:1>) → Wd<14:0>, (Ws<0>) → Wd<15>  
                               For byte operation:  
                               (Ws<7:1>) → Wd<6:0>,        (Ws<0>) → Wd<7>



**Status Affected:** N, Z

Encoding:	1101	0011	0Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

**Description:** Rotate the contents of the source register Ws one bit to the right and place the result in the destination register Wd. The Carry Flag bit is not affected.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select source address mode 2.
- The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

Example1                      RRNC      W5,W6                      ; Rotate right

### Before Instruction

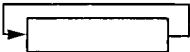
### After Instruction

# RRNCF

Rotate Right f (No Carry)

Syntax: {label:} RRNC{.b} f {,Ww}

Operands: f ∈ [0 ... 8191]  
Operation: For word operation:  
(f<15:1>) → Dest<14:0>, (f<0>) → Dest<15>  
For byte operation:  
(f<7:1>) → Dest<6:0>, (f<0>) → Dest<7>



Status Affected:	N, Z					
Encoding:	1101	0111	0BDf	ffff	ffff	ffff

Description: Rotate the contents of the file register f one bit to the and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register. The carry flag bit is not affected.

The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination.  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 RRNC RAM135, Ww ; Rotate right

Before Instruction

After Instruction



# SCRATCH

## Push Shadow Registers

**Syntax:** {label:} PUSH.S

Operands:                      None

Operation: Push shadow registers. Shadowed registers include W0...W15 and STATUS.

Status Affected: None

Encoding:

1111	1110	1010	0000	0000	0000
------	------	------	------	------	------

Description:	The contents of the primary registers are copied into the shadow registers.
--------------	---

Words: 1

Cycles: 1

## Examples

**Example1**                      **PUSH.S**                      ; Push registers to shadows

### Before Instruction

### After Instruction

[illegible]



**SE** **Sign Extend Wn**

### Sign Extend Wn

Syntax:	{label:}	SE	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++] ,	[Wd++]
			[Ws--] ,	[Wd--]

Operands:           Ws ∈ [W0 ... W15];  
                      Wd ∈ [W0 ... W15]

```

Operation:      Wd<7:0> → Wd<7:0>;
                  if [Ws<7> = 1] then
                    0xFF → Wd<15:8>
                  else
                    0 → Wd<15:8>;

```

Status Affected: C,N,Z

Encoding:	1111	1011	00qq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

<b>Description:</b>	SE sign-extends the eight bit value in Wn (LSB's) to a 16-bit value.
---------------------	--

The 's' bits select the address of the source register.  
The 'd' bits select the address of the destination register.  
The 'p' bits select source address mode 2.  
The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The operation converts a byte to a word.

Words: 1

Cycles: 1

## Examples

Example1                      SE              W5                      ; Sign extend

### Before Instruction

### After Instruction

[illegible]

SETM		Set Ws
Syntax:	{label:}	SETM{.b}
		Ws
		[Ws]
		[Ws]++
		[Ws]--
		[Ws++]
		[Ws--]

Operation:            0xFFFF → Ws for word operation  
                             0xFF → Ws for byte operation

Encoding:	1110	1011	1B00	0000	0ppp	ssss
-----------	------	------	------	------	------	------

The 'B' bits selects byte or word operation.  
The 's' bits select the address of the source register.  
The 'p' bits select the source address mode 2 (values 0-4).

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

## Examples

Example1                      SETM    W7                      ; Set W7 register

Before Instruction

After Instruction

# SETF

Set or Ww

Syntax: {label:} SETM{.b} f  
Ww

Operands: f ∈ [0 ... 8191]  
Operation: 0xFFFF → destination designated by D  
Status Affected: None

Encoding:	1110	1111	1BDf	fff	fff	fff
-----------	------	------	------	-----	-----	-----

Description: Set the register designated by D: If the optional Ww is specified, D=0 and set Ww; otherwise, D=1 and set the file register.

The 'B' bit selects byte or word operation.  
The 'f' bits select the address of the file register.  
The 'D' bit selects the destination.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1                      SETM    345                      ; Set location 345  
Before Instruction  
  
After Instruction



# SFTACK

## Arithmetic Shift Accumulator

Syntax: {label:} SFTAC A, Slit5  
B,

Operands: Slit5 ∈ [-16 ... 15]  
 Operation: Shift<sub>k</sub>(ACC)  
 Status Affected: OA, OB, SA, SB  
 Encoding: 

1100	1000	A100	0000	000k	kkkk
------	------	------	------	------	------

  
 Description: Arithmetic shift of accumulator.

The Slit5 is used as the shift amount. If Slit5 is positive, the shift is a right shift by Slit5 bits. If Slit5 is negative, the shift is a left shift by -Slit5 bits.

The 'A' bit selects the accumulator for the result.  
 The 'k' bits determine the number of bits to be shifted.

Words: 1  
 Cycles: 1

### Examples

Example1 SFTAC B,5 ; Shift Accumulator B right five bits

Before Instruction

After Instruction

09876543210

SL

Shift Left Ws

Syntax:	{label:}	SL{.b}	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++] ,	[Wd++]
			[Ws--] ,	[Wd--]

Operands: Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]  
Operation: For word operation:  
(Ws<15>) → C, (Ws<14:0>) → Wd<15:1>, 0 → Wd<0>  
For byte operation:  
(Ws<7>) → C, (Ws<6:0>) → Wd<7:1>, 0 → Wd<0>



Status Affected: C, N, OV, Z

Encoding:	1101	0000	0Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Shift the contents of the source register Ws one bit to the left and place the result in the destination register Wd. Shift '0' into the LSB of Wd. The Carry Flag is set if the MSB of Ws is '1'.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select source address mode 2.
- The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

Examples

Example1 SL W5,W6 ; Shift left  
Before Instruction  
  
After Instruction

# SLEEP

## Enter SLEEP mode

**Syntax:** {label:} SLEEP lit4

Operands:  $\text{lit4} \in [0 \dots 15]$

Operation:

- 0 → WDT,
- 0 → WDT prescaler count,
- 1 →  $\overline{TO}$ ,
- 0 → PD

Enter sleep mode (lit4)

Status Affected: TO, PD

Encoding:	1111	1110	0100	0000	0000	kkkk
-----------	------	------	------	------	------	------

Description:	The power-down status bit, $\overline{\text{PD}}$ is cleared. Time-out status bit, $\overline{\text{TO}}$ is set. The Watchdog Timer and its prescaler are cleared. The processor is put into SLEEP mode selected by lit4.
--------------	--

Words: 1

Cycles: 1

## Examples

<b>Example1</b>	<b>SLEEP</b>	<b>0</b>	<b>; Turn off the device oscillator.</b>
-----------------	--------------	----------	--

### Before Instruction

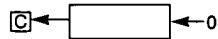
### After Instruction

**SECRET**

**068704-2**

Syntax: {label:} SL{.b} f {,Ww}

Operation: For word operation:  
 $(f<15>) \rightarrow (C), (f<14:0>) \rightarrow \text{Dest}<15:1>, 0 \rightarrow \text{Dest}<0>$   
 For byte operation:  
 $(f<7>) \rightarrow (C), (f<6:0>) \rightarrow \text{Dest}<7:1>, 0 \rightarrow \text{Dest}<0>$



Encoding:	1101	0100	B0Df	ffff	ffff	ffff
-----------	------	------	------	------	------	------

The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination.  
The 'f' bits select the address of the file register.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Cycles: 1

**Example1**                      SL            RAM135, Ww            ; Shift left

### After Instruction



# SLK

## Shift Left by Short Literal

Syntax: {label:} SL Wb, lit5, Wnd

Operands: Wb ∈ [W0 ... W15]; lit5 ∈ [0...31]; Wnd ∈ [W0 ... W15]

Operation: lit5<3:0>→Shift\_Val

0→Shift\_In<39:16>

Wb<15:0>→Shift\_In<15:0>

0→Shift\_Out<39:16+Shift\_Val>

Shift\_In<15:0>→Shift\_Out<15+Shift\_Val:Shift\_Val>

If lit5<4>==0: (less than 16)

0→CARRY1<15:0>

Shift\_Out<31:16>→CARRY0<15:0>

Shift\_Out<15:0>→Wnd<15:0>

If lit5<4>==1: (16 or greater)

Shift\_Out<31:16>→CARRY1<15:0>

Shift\_Out<15:0>→CARRY0<15:0>

0→Wnd<15:0>

Status Affected: C,SZ,Z

Encoding:	1101	1101	0www	wddd	d11k	kkkk
-----------	------	------	------	------	------	------

Description: Shift left the contents of the source register Wb by lit5 bits (up to 31 positions), placing the result in the destination register Wnd. Bits that are shifted beyond the leftmost position of the source are stored in the CARRY1 and CARRY0 registers.

The Z and SZ bits will be set if the value placed in Wnd is zero and cleared otherwise. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.

**Note:** This instruction operates in word mode only.

Words: 1

Cycles: 1

### EXAMPLES:

# SLW Shift Left by Wns

Syntax: {label:} SL Wb, Wns, Wnd

Operands: Wb ∈ [W0 ... W15]; Wns ∈ [W0 ...W15]; Wnd ∈ [W0 ... W15]  
 Operation: Wns<3:0>→Shift\_Val

0→Shift\_In<39:16>  
 Wb<15:0>→Shift\_In<15:0>  
  
 0→Shift\_Out<39:16+Shift\_Val>  
 Shift\_In<15:0>→Shift\_Out<15+Shift\_Val:Shift\_Val>  
  
 If Wns<4>==0: (less than 16)  
     0→CARRY1<15:0>  
     Shift\_Out<31:16>→CARRY0<15:0>  
     Shift\_Out<15:0>→Wnd<15:0>  
 If Wns<4>==1: (16 or greater)  
     Shift\_Out<31:16>→CARRY1<15:0>  
     Shift\_Out<15:0>→CARRY0<15:0>  
     0→Wnd<15:0>

Status Affected: C,SZ,Z  
 Encoding:
 

1101	1101	0www	wddd	d000	ssss
------	------	------	------	------	------

Description: Shift left the contents of the source register Wb by Wns bits (up to 31 positions), placing the result in the destination register Wnd. Bits that are shifted beyond the leftmost position of the source are stored in the CARRY1 and CARRY0 registers.  
  
 The Z and SZ bits will be set if the value placed in Wnd is zero and cleared otherwise. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.

**Note:** This instruction operates in word mode only.

Words: 1  
 Cycles: 1

## EXAMPLES:

**SECRET**

Syntax:	{label:} MPY	A, Wm*Wm	,Wxp,[Wx]	,Wyp,[Wy]
		B,	,Wxp,[Wx]+=kx	,Wyp,[Wy]+=ky
			,Wxp,[Wx]-=kx ‡	,Wyp,[Wy]-=ky ‡
			,Wxp,[W5+W8]	,Wyp,[W7+W8]
			<i>none</i>	<i>none</i>

**Operands:**  $W_m * W_m \in \{W_0 * W_0; W_1 * W_1; W_2 * W_2; W_3 * W_3\}$   
 $W_{xp} \in \{W_0 \dots W_3\}; W_x \in \{W_4, W_5\}; k_x \in \{-6, -4, -2, 2, 4, 6\};$   
 $W_{yp} \in \{W_0 \dots W_3\}; W_y \in \{W_6, W_7\}; k_y \in \{-6, -4, -2, 2, 4, 6\};$   
**Operation:**  $(W_m) * (W_m) \rightarrow ACC(A \text{ or } B);$   
 $([W_x]) \rightarrow W_{xp}; (W_x) + k_x \rightarrow W_x;$   
 $([W_y]) \rightarrow W_{yp}; (W_y) + k_y \rightarrow W_y;$

Encoding:	1111	00mm	A0xx	yyii	iijj	jj11
-----------	------	------	------	------	------	------

The 'm' bits select the operand register Wm for the square:  
The 'A' bit selects the accumulator for the result. The other accumulator is used for write back.  
The 'i' bits select the Wx pre-fetch operation.  
The 'j' bits select the Wy pre-fetch operation.  
The 'x' bits select the pre-fetch Wxp destination.  
The 'y' bits select the pre-fetch Wyp destination.  
The 'a' bits select the accumulator write-back destination.

See Table 1-9 through Table 1-14 for modifier addressing information.

Words: 1  
Cycles: 1

## Examples

Example1    MPY    A,W2\*W2,W0=[W4]-=6,W1=[W6]    ; Square to accumulator A

Before Instruction

ACCA = 2

ACCB = 3

W0 = 5

W1 = 6

W2 = 7

W3 = 8

W8 = 1000

W10 = 2000

RAM(994) = 16

RAM(1000) = 17

RAM(2000) = 18

After Instruction

ACCA =  $2+7*8=58$

ACCB = 3

W0 = 17

W1 = 18

W2 = 7

W3 = 8

W8 = 994

W10 = 2000

RAM(994) = 3

RAM(1000) = 17

RAM(2000) = 18

060401 "050457 25402860

SQRAC

Square and Accumulate

Syntax:	{label;} MAC	A, Wm*Wm	,Wxp,[Wx]	,Wyp,[Wy]	,AWB
	B,		,Wxp,[Wx]+=kx	,Wyp,[Wy]+=ky	none
			,Wxp,[Wx]-=kx ‡	,Wyp,[Wy]-=ky ‡	
			,Wxp,[W5+W8]	,Wyp,[W7+W8]	
			none	none	

‡ Alternate format for negative kx,ky

Operands: Wm\*Wm ∈ {W0\*W0; W1\*W1; W2\*W2; W3\*W3}  
Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6};  
Wyp ∈ {W0 ... W3}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6};  
AWB ∈ {W9, [W9]++}

Operation: (ACC(A or B)) + (Wm)\*(Wm) → ACC(A or B);  
([Wx])→ Wxp; (Wx)+kx→Wx;  
([Wy])→ Wyp; (Wy)+ky→Wy;  
(ACC(B or A)) rounded → AWB

Status Affected: OA, OB, SA, SB

Encoding:	1111	00mm	A0xx	yyii	iijj	jjaa
-----------	------	------	------	------	------	------

Description: Signed, fractional or integer square the contents of a W register. The 32-bit result is sign-extended to 40-bits and added to the specified accumulator. Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required. Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required. AWB specifies the direct or indirect store of the convergently rounded contents of other accumulator, if required.

The 'm' bits select the operand register Wm for the square:  
The 'A' bit selects the accumulator for the result. The other accumulator is used for write back.  
The 'i' bits select the Wx pre-fetch operation.  
The 'j' bits select the Wy pre-fetch operation.  
The 'x' bits select the pre-fetch Wxp destination.  
The 'y' bits select the pre-fetch Wyp destination.  
The 'a' bits select the accumulator write-back destination.

See Table 1-9 through Table 1-14 for modifier addressing information.

Words: 1  
Cycles: 1

## Examples

Example1    MAC    A,W2\*W2,W0=[W4]-=6,W1=[W6],[W9]++ ; Square and Accumulate A

Before Instruction

ACCA = 2

ACCB = 3

W0 = 5

W1 = 6

W2 = 7

W3 = 8

W8 = 1000

W10 = 2000

RAM(994) = 16

RAM(1000) = 17

RAM(2000) = 18

After Instruction

ACCA =  $2 + 7 * 8 = 58$

ACCB = 3

W0 = 17

W1 = 18

W2 = 7

W3 = 8

W8 = 994

W10 = 2000

RAM(994) = 3

RAM(1000) = 17

RAM(2000) = 18

FOR90725402860

SRAC

Store Rounded Accumulator

Syntax:	{label:}	SAC.R	A,	Wnd,	[, Slit4]
			B,	[Wnd],	
				[Wnd]++	
				[Wnd]--	
				[Wnd--],	
				[Wnd+Wb],	
				[Wnd+lit5]	

Operands:	W <sub>nd</sub> ∈ [W0 ... W15]; W <sub>b</sub> ∈ [W0 ... W15]; lit5 ∈ [0 ... 31] Slit4 ∈ [-8 ... +7]					
Operation:	Shift <sub>Slit4</sub> (ACC) (optional);Round(ACC);(ACC[31:16]) → W <sub>nd</sub>					
Status Affected:	None					
Encoding:	1100	1101	Awww	wrrr	rhhh	ssss
Description:	Optionally shift accumulator, round and store convergent rounded accumulator, ACC, to the destination effective address.					

The 'A' bits specify the source accumulator.  
The 's' bits specify the destination register Wnd.  
The 'h' bits select destination address mode 3.  
The 'w' bits specify the offset amount lit5 OR the offset register Wb.  
The 'r' bits encode the optional operand Slit4 which determines the amount of the accumulator preshift; if the operand Slit4 is absent, a 0 is encoded.

See Table 1-7 for modifier addressing information.

**Note:** Positive values of operand Slit4 represent arithmetic shift right.  
Negative values of operand Slit4 represent shift left.

Words:	1
Cycles:	1

Examples

Example1	SAC.R	B,W5	; Store RoundedAccumulator
	Before Instruction		
	After Instruction		

# STDW

### Double Word Move from Wns to Stack or destination

Syntax:	{label:}	MOV.D	Wns	,Wd
				,[Wd]
				,[Wd]++
				,[Wd]--
				,[Wd++]
				,[Wd--]
		PUSH.D	Wns	

Operands:  $Wns \in [W0 \dots W14]$   
 $Wd \in [W0 \dots W15]$

**Operation:** See Section 5.6

Status Affected:           None

Encoding:	1011	1110	10qq	qddd	d000	sss0
-----------	------	------	------	------	------	------

**Description:** This instruction moves two registers to two other locations.in one cycle.

First move a specified Wns register to [Wd] and update Wd according to the addressing mode, then move the next higher adjacent Wns register to [Wd] and update Wd.

The assembly mnemonic `PUSH.D` translates to `MOV.D Wns,[W15]--`

The 's' bits select the address of the source register pair.

The 'd' bits select the address of the destination register.

The 'q' bits select destination address mode 2.

See Table 1-6 for modifier addressing information.

**Note:** This instruction only operates on double word operands

Words: 1

Cycles: 2

## Examples

**Example1**                      PUSH.D    W4                      ; Push W4 and W5 into stack

### Before Instruction

### After Instruction



# STQW

### Quad Word Move from Wns to Stack or destination

Syntax:	{label:}	MOV.Q	Wns	,[Wd]
				,[Wd]++
				,[Wd]--
				,[Wd++]
		PUSH.Q	Wns	

Operands:  $Wns \in [W0, W4, W8, W12]$   
 $Wd \in [W0 \dots W15]$

Operation: See Section 5.6

Status Affected:           None

Encoding:	1011	1110	11qq	qddd	d000	ss00
-----------	------	------	------	------	------	------

Description:	This instruction supports fast context switch by storing four registers in one cycle.
--------------	---

The assembly mnemonic “PUSH.Q Wns” translates to MOV.Q Wns,[W15]--

The 's' bits select the address of the source register quad.

The 'd' bits select the address of the destination register.

The 'q' bits select destination address mode 2.

See Table 1-6 for modifier addressing information.

**Note:** This instruction only operates on quad word operands

**Words:** 1

Cycles: 1

## Examples

**Example1**                      PUSH.Q    W4                      ; Push W4,W5,W6,W7 into stack

### Before Instruction

### After Instruction

# STW

Move Wn to f

Syntax:	{label:}	MOV	Wn,	f		
Operands:	f ∈ [0 ... 65535] Wn ∈ [W0 ... W15]					
Operation:	(Wn) → f					
Status Affected:	None					
Encoding:	1001	ssss	ffff	ffff	ffff	ffff
Description:	Move the contents of a specified W register to any file register.					

The 's' bits select the address of the source register.  
The 'f' bits select the address of the file register.

**Note:** This instruction only operates on word operands

Words:	1
Cycles:	1

## Examples

Example1	MOV    W6,RAM100    ; Move W6 to RAM100
	Before Instruction
	After Instruction

# SUB

Subtract Ws from Wb

Syntax:	{label:}	SUB{.b}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws]++,	[Wd]++
				[Ws]--,	[Wd]--
				[Ws++] ,	[Wd++]
				[Ws--],	[Wd--]

Operands: Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]  
Operation: (Wb) - (Ws) → Wd  
Status Affected: C, DC, N, OV, Z

Encoding:	0101	0www	wBqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Subtract the contents of the source register Ws from the contents of the base register Wb and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.  
The 's' bits select the address of the source register.  
The 'w' bits select the address of the base register.  
The 'd' bits select the address of the destination register.  
The 'p' bits select source address mode 2.  
The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1                      SUB      W5,W6,W7                      ; Subtract W5 from W6  
Before Instruction  
  
After Instruction

# SUBAB

## Subtract Accumulators

Syntax:	{label:}	SUB	A
			B

Operands: none

Operation:           if (SUBAB A) then ACCA - ACCB  $\rightarrow$  ACCA  
                          if (SUBAB B) then ACCB - ACCA  $\rightarrow$  ACCB

**Status Affected:** OA, OB, SA, SB

**Encoding:**

1100	1011	A011	0000	0000	0000
------	------	------	------	------	------

**Description:**

Subtract Accumulators and write results to selected accumulator.

The 'A' bits specify the destination accumulator.

**Words: 1**

Cycles: 1

## Exàmples

Example1	SUB	B	; Subtract ACCA from ACCB, result to ACCB
----------	-----	---	---

### Before Instruction

### After Instruction

05060109

# SUBBFW

Subtract f and Carry bit from Ww

Syntax: {label:} SUBRB{.b} f {,Ww}

Operands: f ∈ [0 ... 8191]  
Operation: (Ww) - (f) - ( $\overline{C}$ ) → destination designated by D  
Status Affected: C, DC, N, OV, Z  
Encoding: 

1011	1101	1BDf	ffff	ffff	ffff
------	------	------	------	------	------

  
Description: Subtract the contents of the file register and the carry bit from the contents of the working register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.  
  
The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination.  
The 'f' bits select the address of the file register.  
  
**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 SUBRB RAM135, Ww ; Subtract  
Before Instruction  
  
After Instruction

SUBB

Subtract Ws from Wb with Borrow

Syntax:	{label:}	SUBB{.b}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws]++,	[Wd]++
				[Ws]--,	[Wd]--
				[Ws++],	[Wd++]
				[Ws--],	[Wd--]

Operands: Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]

Operation: (Wb) - (Ws) - (C̄) → Wd

Status Affected: C, DC, N, OV, Z

Encoding:	0101	1www	wBqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Subtract the contents of the source register Ws and the Carry flag from the contents of the base register Wb and place the result in the destination register Wd.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source register.
- The 'w' bits select the address of the base register.
- The 'd' bits select the address of the destination register.
- The 'p' bits select source address mode 2.
- The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

Examples

Example1                      SUBB    W5,W6,W7                      ; Subtract  
Before Instruction  
  
After Instruction

09870457 "060101

# SUBBLS

Subtract Short Literal from Wb with Borrow

Syntax:	{label:}	SUBB{.b}	Wb,	lit5,	Wd
					[Wd]
					[Wd]++
					[Wd]--
					[Wd++]
					[Wd--]

Operands: Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]

Operation: (Wb) - lit5 - (C̄) → Wd

Status Affected: C, DC, N, OV, Z

Encoding:	0101	1www	wBqq	qddd	d11k	kkkk
-----------	------	------	------	------	------	------

Description: Subtract the literal operand and the Carry bit from the contents of the base register Wb and place the result in the destination register Wd.

- The 'B' bit selects byte or word operation.
- The 'w' bits select the address of the base register.
- The 'k' bits provide the literal operand, a five-bit integer number.
- The 'd' bits select the address of the destination register.
- The 'q' bits select destination address mode 2.

See Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1                      SUBB    W5,#12,W7                      ; Subtract  
Before Instruction  
  
After Instruction

# SUBBLW

Subtract Wn from Literal with Borrow

Syntax: {label:} SUBB{.b} Slit10, Wn

Operands: Slit10 ∈ [-512 ... 511]; Wn ∈ [W0 ... W15]  
Operation:  $Slit10 - (Wn) - (\overline{C}) \rightarrow Wn$   
Status Affected: C, DC, N, OV, Z  
Encoding: 

1011	0001	1Bkk	kkkk	kkkk	dddd
------	------	------	------	------	------

  
Description: Subtract the literal operand and the Carry bit from the contents of the working register Wn and place the result in the working register Wn.

The 'B' bit selects byte or word operation.  
The 'd' bits select the address of the working register.  
The 'k' bits specify the literal operand, a signed 10-bit number.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 SUBB #123,W7 ; Subtract  
Before Instruction  
  
After Instruction



### Subtract Wb from Ws with Borrow

Syntax:	{label:}	SUBBR{.b}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws]++,	[Wd]++
				[Ws]--,	[Wd]--
				[Ws+++],	[Wd+++]
				[Ws--],	[Wd--]

Operands:  $Wb \in [W0 \dots W15]; Ws \in [W0 \dots W15]; Wd \in [W0 \dots W15]$

Operation:  $(Ws) - (Wb) - (\bar{C}) \rightarrow Wd$

**Status Affected:** C, DC, N, OV, Z

**Encoding:**

0001	1www	wBqq	qddd	dppp	ssss
------	------	------	------	------	------

**Description:**

Subtract the contents of the base register Wsb and the Carry flag from the contents of the source register Ws and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.

The 's' bits select the address of the source register.

The 'w' bits select the address of the base register.

The 'd' bits select the address of the destination register.

The 'p' bits select source address mode 2.

The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

**Example1**                      SUBBR    W5,W6,W7                      ; Subtract W6 from W5 with borrow

### Before Instruction

### After Instruction

# SUBBRLS

Subtract Wb from Short Literal with Borrow

Syntax: {label:} SUBBR{.b} Wb, lit5 Wd  
[Wd]  
[Wd]++  
[Wd]--  
[Wd++]  
[Wd--]

Operands: Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]

Operation: lit5 - (Wb) - ( $\overline{C}$ ) → Wd

Status Affected: C, DC, N, OV, Z

Encoding:

0001	1www	wBqq	qddd	d11k	kkkk
------	------	------	------	------	------

Description: Subtract the contents of the base register Wb and the Carry flag from lit5 and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.

The 'w' bits select the address of the base register.

The 'k' bits provide the literal operand, a five-bit integer number.

The 'd' bits select the address of the destination register.

The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

Example1 SUBBR W5,#12,W7 ; Subtract W5 from 12

Before Instruction

After Instruction

## SUBBWF

**Subtract Ww and Carry bit from f**

Syntax: {label:} SUBB{.b} f {,Ww}

Operands:  $f \in [0 \dots 8191]$

Operation: (f) - (Ww) - ( $\bar{C}$ ) → destination designated by D

**Status Affected:** C, DC, N, OV, Z

Encoding:

1011	0101	1BDf	ffff	ffff	ffff
------	------	------	------	------	------

**Description:**

Subtract the contents of the working register and the carry bit from the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.

The 'D' bit selects the destination.

The 'f' bits select the address of the file register.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

**Example1**                      SUBB        RAM135, Ww        ; Subtract

### Before Instruction

### After Instruction

[illegible]

# SUBFW

Subtract f from Ww

Syntax: {label:} SUBR{.b} f {,Ww}

Operands: f ∈ [0 ... 8191]  
Operation: (Ww) - (f) → destination designated by D  
Status Affected: C, DC, N, OV, Z

Encoding:	1011	1101	0BDf	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description: Subtract the contents of the file register from the contents of the working register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.  
The 'D' bit selects the destination.  
The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 SUBR RAM135, ww ; Subtract  
Before Instruction  
  
After Instruction

# SUBLS

Subtract Short Literal from Wb

Syntax:	{label:}	SUB{.b}	Wb,	lit5,	Wd
					[Wd]
					[Wd]++
					[Wd]--
					[Wd++]
					[Wd--]

Operands: Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]  
Operation: (Wb) - lit5 → Wd  
Status Affected: C, DC, N, OV, Z

Encoding:	0101	0www	wBqq	qddd	d11k	kkkk
-----------	------	------	------	------	------	------

Description: Subtract the literal operand from the contents of the base register Wb and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.  
The 'w' bits select the address of the base register.  
The 'k' bits provide the literal operand, a five-bit integer number.  
The 'd' bits select the address of the destination register.  
The 'q' bits select destination address mode 2.

See Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1                      SUB      W5,#12,W7                      ; Subtract  
Before Instruction  
  
After Instruction

### Subtract Wn from Literal

Syntax: {label;} SUB{.b} Slit10, Wn

**Operands:** Slit10  $\in [-512 \dots 511]$ ; Wn  $\in [W0 \dots W15]$

Operation:  $\text{Slit10} - (W_n) \rightarrow W_n$

**Status Affected:** C, DC, N, OV, Z

**Encoding:**

1011	0001	0Bkk	kkkk	kkkk	dddd
------	------	------	------	------	------

**Description:**

Subtract the working register from the contents of the literal operand and place the result in the working register Wn.

The 'B' bit selects byte or word operation.

The 'd' bits select the address of the working register.

The 'k' bits specify the literal operand, a signed 10-bit number.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

**Example1**                      SUB     #123,W7                      ; Subtract

### Before Instruction

### After Instruction

# SUBR

Subtract Wb from Ws

Syntax:	{label:}	SUBR{.b}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws]++,	[Wd]++
				[Ws]--,	[Wd]--
				[Ws++] ,	[Wd++]
				[Ws--],	[Wd--]

Operands: Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]  
Operation: (Ws) - (Wb) → Wd  
Status Affected: C, DC, N, OV, Z

Encoding:	0001	0www	wBqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: Subtract the contents of the base register Wb from the contents of the source register Ws and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.  
The 's' bits select the address of the source register.  
The 'w' bits select the address of the base register.  
The 'd' bits select the address of the destination register.  
The 'p' bits select source address mode 2.  
The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 SUBR W5,W6,W7 ; Subtract W6 from W5  
Before Instruction  
  
After Instruction

09870457 060104

# SUBRLS

Subtract Wb from Short Literal

Syntax:	{label:}	SUBR{.b}	Wb,	lit5	Wd
					[Wd]
					[Wd]++
					[Wd]--
					[Wd++]
					[Wd--]

Operands: Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]  
Operation: lit5 - (Wb) → Wd  
Status Affected: C, DC, N, OV, Z

Encoding:	0001	0www	wBqq	qddd	d11k	kkkk
-----------	------	------	------	------	------	------

Description: Subtract the contents of the base register Wb from the lit5 and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.  
The 'w' bits select the address of the base register.  
The 'k' bits provide the literal operand, a five-bit integer number.  
The 'd' bits select the address of the destination register.  
The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 SUBR W5,#12,W7 ; Subtract W5 from 12  
Before Instruction  
  
After Instruction



# SUBWF

Subtract Ww from f

Syntax: {label:} SUB{.b} f {,Ww}

Operands: f ∈ [0 ... 8191]  
 Operation: (f) - (Ww) → destination designated by D  
 Status Affected: C, DC, N, OV, Z

Encoding:	1011	0101	0BDF	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description: Subtract the contents of the working register from the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.  
 The 'D' bit selects the destination.  
 The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
 Cycles: 1

## Examples

Example1                      SUB              RAM135, ww              ; Subtract  
                                  Before Instruction  
  
                                  After Instruction

# SWAP

Byte or Nibble Swap Wn

Syntax: {label:} SWAP Wn

Operands:

Wn ∈ [W0 ... W15]

Operation:

If B=0; (Wn)<15:8> ↔ (Wn)<7:0>  
If B=1; (Wn)<7:4> ↔ (Wn)<3:0>

Status Affected:

None

Encoding:

1111	1101	1B00	0000	0000	ssss
------	------	------	------	------	------

Description:

If in word mode, byte swap Wn register.  
If in byte mode, nibble swap Wn register. Wn<15:8> are unaffected.  
  
The 'B' bit selects byte or word operation.  
The 's' bits select the address of the working register.  
  
**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words:

1

Cycles:

1

## Examples

Example1

SWAPW11; Swap Bytes

Before Instruction

After Instruction

## TBLRDH

### Table Read High

Syntax:	{label:}	TBLRDH{.b}	[Ws],	Wd
			[Ws]++,	[Wd]
			[Ws]--,	[Wd]++
			[Ws++] ,	[Wd]--
			[Ws--],	[Wd++]
				[Wd--]

Operands:           Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]

Operation: In Word Mode:

Program Mem [(PAGNUM),(Ws)] <23:16> → Wd <7:0>

$$0 \rightarrow Wd <15:8>$$

**In Byte Mode:**

If  $LSB(Ws)=1$ ,  $0 \rightarrow Wd<7:0>$

Else if  $LSB(Ws)=0$ , Program Mem [(PAGNUM),(Ws)]  $\langle 23:16 \rangle \rightarrow Wd \langle 7:0 \rangle$

Status Affected: None

Encoding:	1011	1010	1Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description:	This instruction is used to read the contents of program memory.
--------------	--

The program memory address is calculated by concatenating the contents of the 8-bit Table Pointer (PAGNUM) register with the contents of the Ws register.

Because the Ws value is always used as an address, the direct form of the first operand is invalid.

The program memory word is stored in the location indicated by the Wd operand.

For this instruction, the upper 8 bits of the program memory word (extended with '0's) are read.

The 'B' bit selects byte or word operation.

The 's' bits select the address of the source (address) register.

The 'd' bits select the address of the destination (data) register.

The 'p' bits select source address mode 2.

The 'q' bits select destination address mode 2.

**Note:** The extension `.b` in the instruction denotes a byte move rather than a word move. You may use a `.w` extension to denote a word move, but it is not required.

Words: 1

Cycles: 2

## Examples

**Example1**                      TBLRDH W5, W6                      ; Read Program Memory High

Before Instruction

After Instruction

09870457-060404

# TBLRDL

## Table Read Low

Syntax:	{label:}	TBLRDL{.b}	[Ws],	Wd
			[Ws]++,	[Wd]
			[Ws]--,	[Wd]++
			[Ws++] ,	[Wd]--
			[Ws--],	[Wd++]
				[Wd--]

Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	In Word Mode: Program Mem [(PAGNUM),(Ws)] <15:0> → Wd In Byte Mode: If LSB(Ws)=1, Program Mem [(PAGNUM),(Ws)] <15:8> → Wd<7:0> Else if LSB(Ws)=0, Program Mem [(PAGNUM),(Ws)] <7:0> → Wd<7:0>					
Status Affected:	None					
Encoding:	1011	1010	0Bqq	qddd	dppp	ssss
Description:	This instruction is used to read the contents of program memory.					

The program memory address is calculated by concatenating the contents of the 8-bit Table Pointer (PAGNUM) register with the contents of the Ws register.

Because the Ws value is always used as an address, the direct form of the first operand is invalid.

The program memory word is stored in the location indicated by the Wd operand.

For this instruction, the lower 16 bits of the program memory word are read.

- The 'B' bit selects byte or word operation.
- The 's' bits select the address of the source (address) register.
- The 'd' bits select the address of the destination (data) register.
- The 'p' bits select source address mode 2.
- The 'q' bits select destination address mode 2.

**Note:** The extension .b in the instruction denotes a byte move rather than a word move. You may use a .w extension to denote a word move, but it is not required.

Words:	1
Cycles:	2

### Examples

Example1	TBLRDL W5, W6	; Read Program Mememory Low
----------	---------------	-----------------------------

Before Instruction

After Instruction

09270457.060104

# TBLWTH

## Table Write High

Syntax:	{label:}	TBLWTH	Ws,	[Wd]
			[Ws],	[Wd]++
			[Ws]++,	[Wd]--
			[Ws]--,	[Wd++]
			[Ws++] ,	[Wd--],
			[Ws--],	

Operands: Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]  
 Operation: In Word Mode:  
               (Ws)<7:0>→ Program Mem [(PAGNUM),(Wd)] <23:16>  
               In Byte Mode:  
                   If LSB(Wd)=1, NOP  
                   Else if LSB(Wd)=0, Ws<7:0>→ Program Mem [(PAGNUM),(Wd)]<23:16>

Status Affected: None

Encoding:	1011	1011	1Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: This instruction is used to write the contents of Program Memory.

The program memory address is calculated by concatenating the contents of the 8-bit Table Pointer (PAGNUM) register with the result of the Wd operand.

Because the Wd value is always used as an address, the direct form of the second operand is invalid.

The contents of the Ws operand are stored into program memory at the location indicated by the Wd operand.

This instruction writes the upper 8 bits of the program memory word.

The 'B' bit selects byte or word operation.  
 The 's' bits select the address of the source (data) register.  
 The 'd' bits select the address of the destination (address) register.  
 The 'p' bits select source address mode 2.  
 The 'q' bits select destination address mode 2.

**Note:** The extension .b in the instruction denotes a byte move rather than a word move. You may use a .w extension to denote a word move, but it is not required.

Words: 1  
 Cycles: 2

### Examples

Example1                               TBL-       W5, W6                       ; Load Program Memory High  
   WTH

Before Instruction

After Instruction

09870457"060101



### Table Write Low

Syntax:	{label:}	TBLWTL{.b}	Ws,	[Wd]
			[Ws],	[Wd]++
			[Ws]++,	[Wd]--
			[Ws]--,	[Wd++]
			[Ws++] ,	[Wd--],
			[Ws--],	

**Operands:**  $Ws \in [W0 \dots W15]$ ;  $Wd \in [W0 \dots W15]$   
 $S \in [0, 1]$  (default = 0)

Operation:

In Word Mode:  
(Ws) → Program Mem [(PAGNUM),(Wd)] <15:0>

In Byte Mode:  
If LSB(Ws)=1, Ws<7:0> → Program Mem [(PAGNUM),(Wd)] <15:8>  
Else if LSB(Wd)=0, Ws<7:0> → Program Mem [(PAGNUM),(Wd)] <7:0>

Status Affected: None

Encoding:	1011	1011	0Bqq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

<b>Description:</b>	This instruction is used to program the contents of Program Memory.
---------------------	---

The program memory address is calculated by concatenating the contents of the 8-bit Table Pointer (PAGNUM) register with the result of the Wd operand.

Because the Wd value is always used as an address, the direct form of the second operand is invalid.

The contents of the Ws operand are stored into program memory at the location indicated by the Wd operand.

This instruction writes the lower 16 bits of the program memory word.

The 'B' bit selects byte or word operation.

The 's' bits select the address of the source (data) register.

The 'd' bits select the address of the destination (address) register.

The 'p' bits select source address mode 2.

The 'q' bits select destination address mode 2

**Note:** The extension `.b` in the instruction denotes a byte move rather than a word move. You may use a `.w` extension to denote a word move, but it is not required.

Words: 1

Cycles: 2

## Examples

**Example1**                      **TBLWTL W5, W6**                      **; Load Program Memory Low**

Before Instruction

After Instruction

09970457 060101

## De-allocate Stack Frame

Syntax:	{label:}	ULNK
---------	----------	------

Operands:                      None

Operation:            W14→ W15;  
                              [W15++] → W14

Status Affected:           None

Encoding:	1111	1010	1000	0000	0000	0000
-----------	------	------	------	------	------	------

Description:	This instruction de-allocates a stack frame and adjusts the stack pointer and frame pointer.
--------------	--

Words: 1

Cycles: 1

## Examples

```
Example1          ULNK          ;Deallocate stack frame
```

**0697018**

### Trap to vector(lit1) with lit16

**Syntax:** {label:} TRAP lit1, lit16

Operands:	lit1 $\in$ [0,1]; lit16 $\in$ [0 ... 65535]
Operation:	(PC) +2 $\rightarrow$ PC, (PC<15:0>) $\rightarrow$ TOS, (W15)+2 $\rightarrow$ W15 (PC<23:16>) $\rightarrow$ TOS, (W15)+2 $\rightarrow$ W15 Vector(lit1) $\rightarrow$ PC; lit16 $\rightarrow$ TOS

Status Affected:           None

**Encoding:**

0000	101n	kkkk	kkkk	kkkk	kkkk
------	------	------	------	------	------

**Description:**

This instruction allows instruction expansion. The instruction will call a vector location with the lit16 value pushed onto the stack.

**Words:** 1

Cycles: 2

## Examples

Example1                      TRAP    #0,#0x5A5A

### Exclusive or Wb and Ws

Syntax:	{label:}	XOR{.b}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws]++,	[Wd]++
				[Ws]--,	[Wd]--
				[Ws++] ,	[Wd++]
				[Ws--] ,	[Wd--]

Operands:  $Wb \in [W0 \dots W15]; Ws \in [W0 \dots W15]; Wd \in [W0 \dots W15]$

Operation:  $(Wb).XOR.(Ws) \rightarrow Wd$

Status Affected: N, Z

**Encoding:**

0110	1www	wBqq	qddd	dppp	ssss
------	------	------	------	------	------

**Description:**

Compute Exclusive OR of the contents of the source register Ws and the contents of the base register4 Wb and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.

The 's' bits select the address of the source register.

The 'w' bits select the address of the base register.

The 'd' bits select the address of the destination register.

The 'p' bits select source address mode 2.

The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

**Example1** XOR W5,W6,W7 ; Xor

### Before Instruction

### After Instruction

[illegible]

### Exclusive Or Wb and Short Literal

Syntax:	{label:}	XOR{.b}	Wb,	lit5,	Wd
					[Wd]
					[Wd]++
					[Wd]--
					[Wd++]
					[Wd--]

Operands:            Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]

Operation:  $(Wb).XOR.lit5 \rightarrow Wd$

Status Affected: N, Z

**Encoding:**

0110	1www	wBqq	qddd	d11k	kkkk
------	------	------	------	------	------

**Description:**

Compute the Exclusive Or of the contents of the base register Wb and the literal operand and place the result in the destination register Wd.

The 'B' bit selects byte or word operation.

The 'w' bits select the address of the base register.

The 'k' bits provide the literal operand, a five-bit integer number.

The 'd' bits select the address of the destination register.

The 'q' bits select destination address mode 2.

See Table 1-6 for modifier addressing information.

**Note:** The extension `.b` in the instruction denotes a byte operation rather than a word operation. You may use a `.w` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

## Examples

**Example1**                      XOR      W5,#12,W7                      ; Exclusive Or

### Before Instruction

### After Instruction

**DECEMBER 1964**

# XORLW

Exclusive Or Literal and Wn

Syntax: {label:} XOR{.b} Slit10, Wn

Operands: Slit10 ∈ [-512 ... 511]; Wn ∈ [W0 ... W15]  
Operation: Slit10.XOR.(Wn) → Wn  
Status Affected: N, Z

Encoding:	1011	0010	1Bkk	kkkk	kkkk	dddd
-----------	------	------	------	------	------	------

Description: Compute the Exclusive Or of the literal operand and the contents of the working register Wn and place the result in the working register Wn.

The 'B' bit selects byte or word operation.  
The 'd' bits select the address of the working register.  
The 'k' bits specify the literal operand, a signed 10-bit number.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1

## Examples

Example1 XOR #123,W7 ; Exclusive Or  
Before Instruction  
  
After Instruction

# XORWF

Exclusive Or f and Ww

Syntax: {label:} ADD{.b} f {,Ww}

Operands: f ∈ [0 ... 8191]  
 Operation: (f).XOR.(Ww) → destination designated by D  
 Status Affected: N, Z

Encoding:	1011	0110	1BDf	ffff	ffff	ffff
-----------	------	------	------	------	------	------

Description: Compute the XOR of the contents of the working register and the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.

The 'B' bit selects byte or word operation.  
 The 'D' bit selects the destination.  
 The 'f' bits select the address of the file register.

**Note:** The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

Words: 1  
 Cycles: 1

## Examples

Example1 XOR RAM135, Ww ; Exclusive Or  
 Before Instruction  
 After Instruction



ZE

Zero Extend Wn

Syntax:	{label:}	ZE	Ws,	Wd
			[Ws],	[Wd]
			[Ws]++,	[Wd]++
			[Ws]--,	[Wd]--
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]

Operands: Ws ∈ [W0 ... W15];  
Wd ∈ [W0 ... W15]

Operation: Ws<7:0> → Wd<7:0>;  
0 → Wd<15:8>;

Status Affected: None

Encoding:	1111	1011	10qq	qddd	dppp	ssss
-----------	------	------	------	------	------	------

Description: ZE zero-extends the eight bit value in Wn (LSB's) to a 16-bit value.

The 's' bits select the address of the source register.  
The 'd' bits select the address of the destination register.  
The 'p' bits select source address mode 2.  
The 'q' bits select destination address mode 2.

See Table 1-5 and Table 1-6 for modifier addressing information.

**Note:** The operation converts a byte to a word.

Words: 1  
Cycles: 1

Examples

Example1                      ZE        W5                      ; Sign extend

Before Instruction

After Instruction

**APPENDIX B**

09870457 .060101

## 4.0 ADDRESS GENERATOR UNITS

address spaces. If they are not, one of the EAs will be outside the address space of the corresponding data space (and will fetch the bus default value, 0x0000).

The dsPIC core contains two independent address generator units. The X AGU is for MCU and DSP instructions. The Y AGU is for DSP MAC class of instructions only. They are capable of supporting three types of data addressing:

- Linear addressing
- Modulo (circular) addressing
- Bit Reversed addressing (X AGU only)

Linear and modulo data addressing modes can be applied to data space or program space. Although bit reversed addressing will work with any EA calculation, by definition it is only applicable to data space.

### 4.0.1 Data Space Organization

- Although the data space memory is organized as 16-bit words, all effective addresses (EAs) point to bytes. Instructions can thus access any byte or aligned words (data words at an even address). Misaligned word accesses are not supported, and if attempted will initiate an address error trap. The LS-bit of the EA is used to determine upper or lower byte access. The LS-bit becomes a 'don't care' for word accesses. Each memory (or register where appropriate) must provide independent upper and lower byte write lines to support byte writes. In addition, a multiplexor must be included to route the LS byte of an operand to the upper or lower byte of the target EA word for both reads and writes.

When executing instructions which require just one source operand to be fetched from data space, the X AGU is used to calculate the effective address. The AGU can generate an address to point to anywhere in the 64K byte data space. It supports all addressing modes, modulo addressing for low overhead circular buffers, and bit reversed addressing to facilitate FFT data reorganization.

- When executing instructions which require two source operands to be concurrently fetched (i.e. the MAC class of DSP instructions), both the X and Y AGUs are used simultaneously and the data space is split into 2 independent address spaces, X and Y. The Y AGU supports register indirect post-modified and modulo addressing only. Note that the data write phase of the MAC class of instruction does not split X and Y address space. The write EA is calculated using the X AGU and the data space is configured for full 64Kbyte access.

- In the split data space mode, some W register address pointers are dedicated to AGU X, others to AGU Y (see Section 1.2.4 for details). The EAs of each operand must therefore be restricted to be within different

## 4.1 Instruction Addressing Modes

The basic set of addressing modes shown in Table 4-1. Note that, 'Wn+= ' indicates that the contents of Wn is added to something to form the effective address which is then written back into Wn. 'Wn+' indicates that the contents of Wn is added to something to form the effective address but the contents of Wn remain unchanged.

The addressing modes in Table 4-1 form the basis of three groups of addressing modes optimized to support specific instruction features. They are MODE1, MODE2 and MODE3. The DSP MAC and derivative instructions are an exception where the addressing modes are encoded differently. This set of addressing modes is referred to as MODE4. Refer to dsPIC Instruction Set DOS for full details.

Addressing Mode	Function	Description
Register Direct	EA = Wn	Wn is the EA
Register Indirect	EA = [Wn]	The contents of Wn forms the EA
Register Indirect Post-modified	EA = [Wn] += 1 EA = [Wn] = 1	The contents of Wn forms the EA which is post-modified by a constant value
Register Indirect Pre-modified	EA = [Wn += 1] EA = [Wn = 1]	Wn is pre-modified by a signed constant value to form the EA
Register Indirect with Register Offset	EA = [Wn + Wb]	The sum of Wn and Wb forms the EA
Register Indirect with Constant Offset	EA = [Wn + constant]	The sum of Wn and a signed constant value forms the EA

**Note 1:** EA = effective address

**2:** All address modification values (except Wb) are scaled for word access

**TABLE 4-1: FUNDAMENTAL ADDRESSING MODES SUPPORTED**

All but a few instructions support both 8-bit and 16-bit operand data sizes. In order to efficiently accommodate this requirement, all effective addresses are byte aligned. As the data space is 16-bits wide, the following consequences must be understood.

1. Mis-aligned word accesses are not supported. All word effective addresses must be even (the LS-bit of the EA is ignored by the data space memory).
2. The LS-bit of the effective address is used to select which byte (upper or lower) is multiplexed onto bits [7:0] of the data bus for byte sized accesses.
3. Post and pre-modification of a register by a constant value to create a new effective address must take into account of the data size accessed. All constant values, whether implied (e.g. post-inc) or declared (e.g. post-modify with S5lit) are scaled by a factor of 2 for word accesses. For example:  
 [Ws] += 1 will post-modify data source pointer Ws by 1 for a byte access, and by 2 for a word access.  
 [Ws] += S5lit5 will post-modify data source pointer Ws by S5lit5 for byte accesses and S5lit5 << 1 (shift left by 1) for word accesses.

**Note:** Register offsets are not scaled.

Unless otherwise noted, it is assumed that all addresses and addressing modes refer to byte size accesses.

**Note:** All addressing modes which have to calculate the EA (pre-modified, register offset and constant offset) have very tight timing requirements which may require some instruction addressing sequence restrictions in future DOS releases.

#### 4.1.1 MODE 1

MODE1 determines the addressing mode for one of the two operand sources required for the three operand instructions (found in categories 'MATH' and 'SKIP'). These instructions are of the form:

Result = Operand 1 <function> Operand 2

Operand1 is always a register (i.e. the addressing mode can only be register direct) which is referred to as Wb. Operand 2 is fetched from data memory based upon the addressing mode selected by MODE1. MODE1 therefore defines one of the source operand addressing modes and implies that of the other source operand.

In addition, MODE1 may also provide a signed 5-bit constant (literal) as the operand. In this case, the instruction is of the form:

Result = Operand 1 <function> signed literal

Operand 1 is always a register (i.e. the addressing mode can only be register direct) which is selected from the Ws field in the instruction. The 4-bit Wb field forms the 4 LS-bits of a signed constant. It is concatenated with the LS-bit of the three bit MODE1 field to form the 5-bit signed constant value.

In summary, MODE1 supports the addressing modes shown in Table 4-2.

MODE1 Bit Encoding	Operand 1		Operand 2	
	Function	Description	Function	Description
000	EA = Wb	Register direct	EA = Ws	Register direct
001	EA = Wb	Register direct	EA = [Ws]	Register indirect
010	EA = Wb	Register direct	EA = [Ws]-= 1	Register indirect post-decremented
011	EA = Wb	Register direct	EA = [Ws]+= 1	Register indirect post-incremented
100	EA = Wb	Register direct	EA = [Ws-=1]	Register indirect pre-decremented
101	EA = Wb	Register direct	EA = [Ws+=1]	Register indirect pre-incremented
110	EA = Ws	Register direct	Operand 2 =	5-bit signed literal
111			S5lit	

TABLE 4-2:MODE1 ADDRESSING MODE DEFINITION

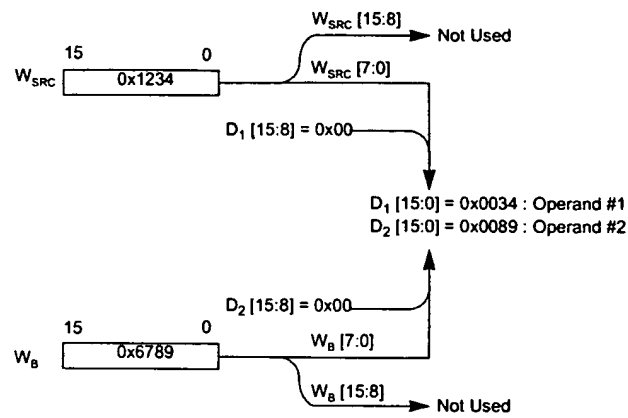
#### 4.1.1.1 Mode1, Register Direct

Addressing MODE1, submode 0 is register direct. The implied effective address is the memory mapped address of register Ws.

Note: Rather than executing a memory fetch, it may be preferable to perform two W-array fetches if bussing allows???

The operand is contained in Ws as shown in Figure 4-1.

##### Byte Operand Size



##### Word Operand Size

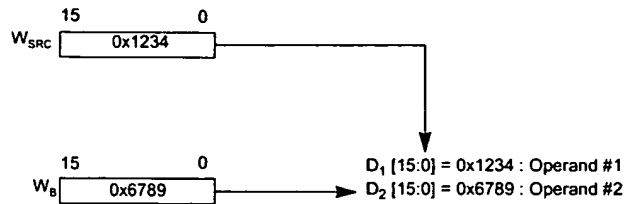
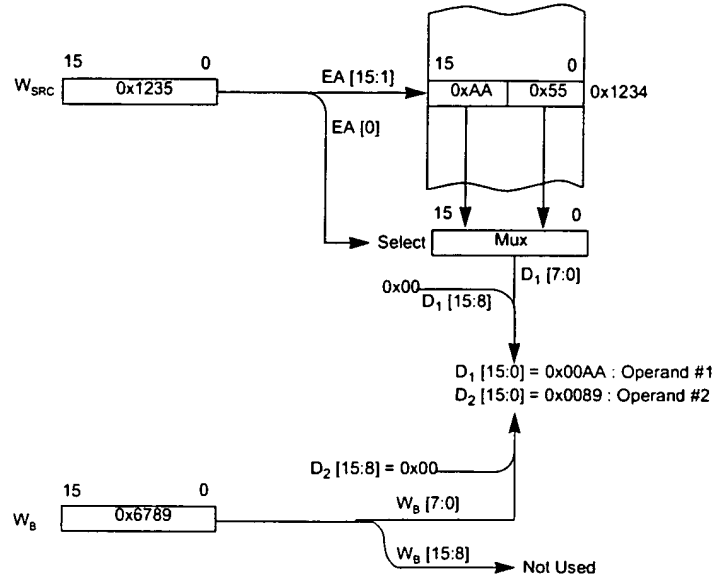


FIGURE 4-1: REGISTER DIRECT (MODE1, SUBMODE 0)

#### 4.1.1.2 Mode1, Register Indirect

Addressing MODE1, submode 1 is register indirect.  
The effective address contained in register Ws points to the operand as shown in Figure 4-2.

##### Byte Operand Size



##### Word Operand Size

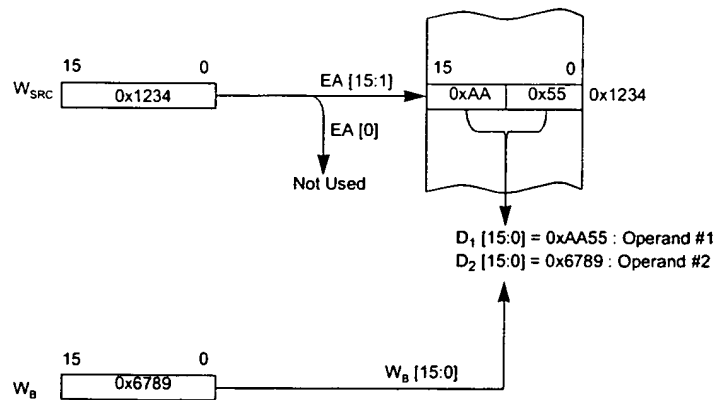


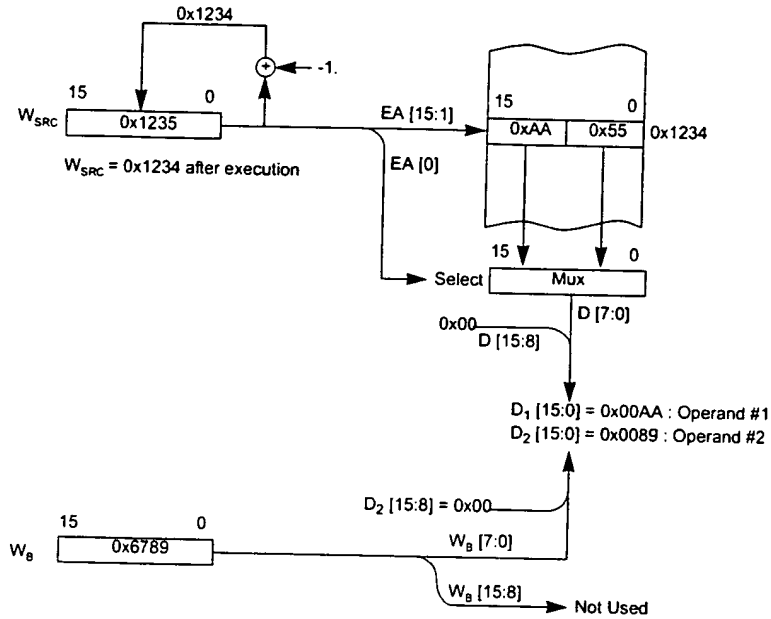
FIGURE 4-2: REGISTER INDIRECT (MODE1, SUBMODE 1)

#### 4.1.1.3 Mode1, Register Indirect with Post Decrement

Ws is then post decremented as shown in Figure 4-3.

Addressing MODE1, submode 2 is register indirect with post decrement. The effective address contained in register Ws points to the operand.

##### Byte Operand Size



##### Word Operand Size

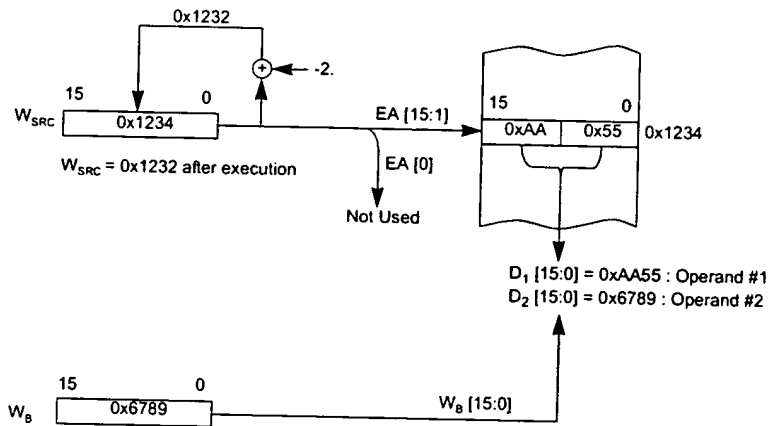


FIGURE 4-3: REGISTER INDIRECT WITH POST DECREMENT (MODE1, SUBMODE 2)

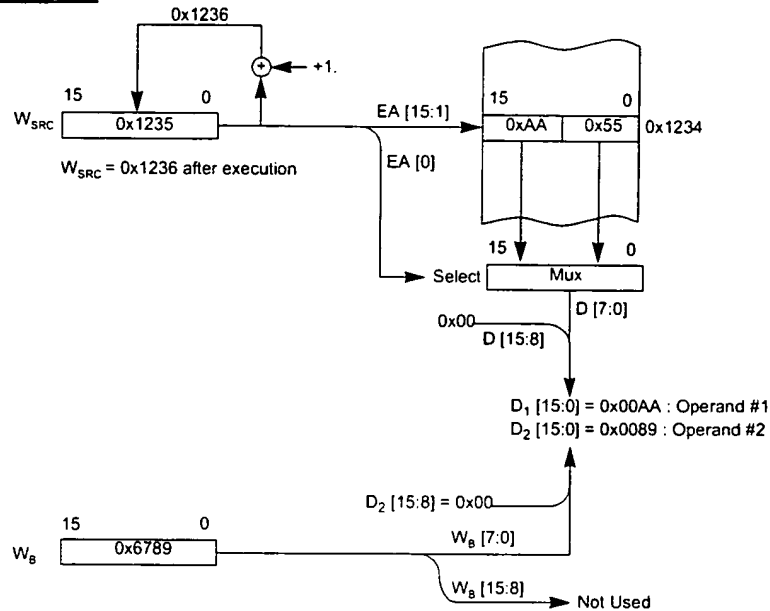


#### 4.1.1.4 Mode1, Register Indirect with Post Increment

Addressing MODE1, submode 3 is register indirect with post increment. The effective address contained in register Ws points to the operand.

Ws is then incremented as shown in Figure 4-4.

##### Byte Operand Size



##### Word Operand Size

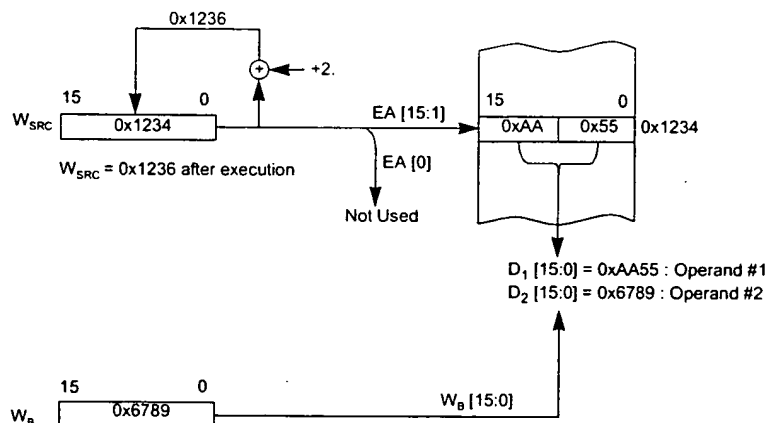


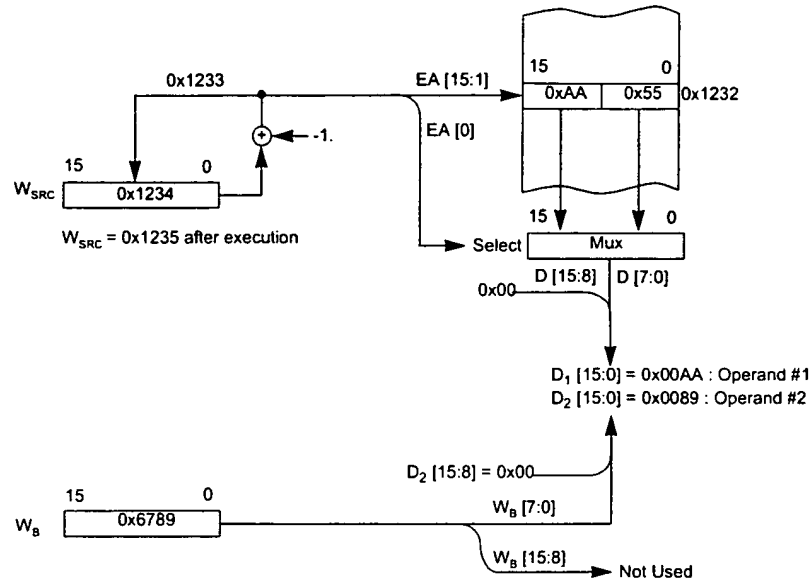
FIGURE 4-4: REGISTER INDIRECT WITH POST INCREMENT (MODE1, SUBMODE 3)

#### 4.1.1.5 Mode1, Register Indirect with Pre Decrement

Addressing MODE1, submode 4 is register indirect with pre-decrement.

Register Ws is decremented to form the effective address which points to the operand as shown in Figure 4-5.

##### Byte Operand Size



##### Word Operand Size

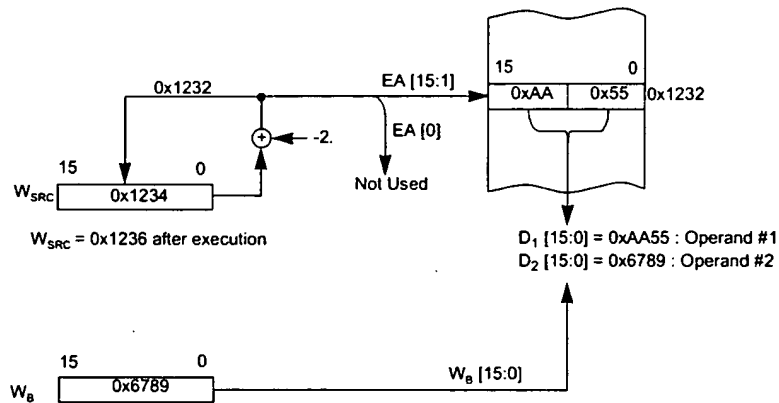


FIGURE 4-5: REGISTER INDIRECT WITH PRE DECREMENT (MODE1, SUBMODE 4)

#### 4.1.1.6 Mode1, Register Indirect with Pre Increment

Addressing MODE1, submode 5 is register indirect with pre increment.

Register Ws is incremented to form the effective address which points to the operand as shown in Figure 4-6.

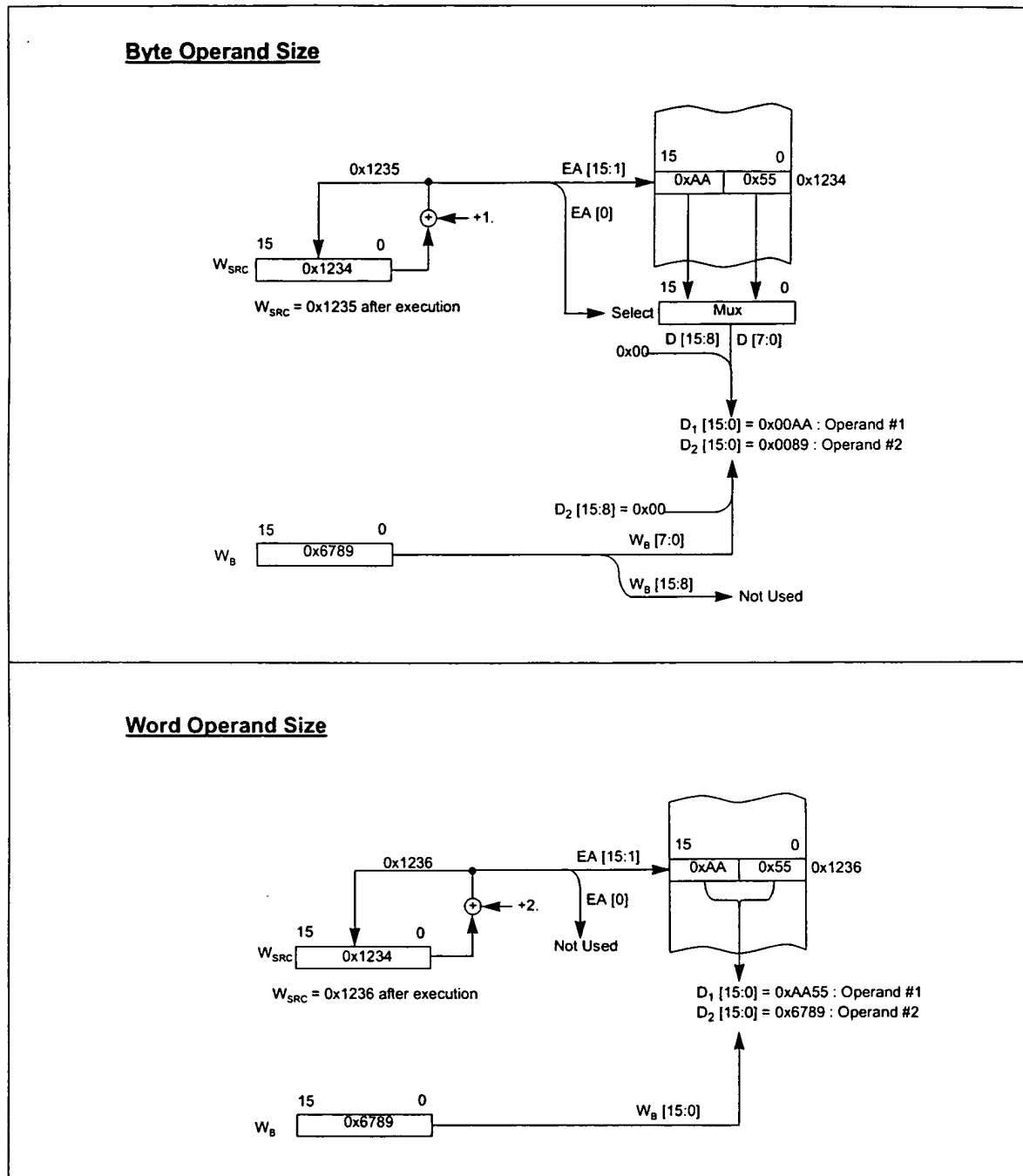


FIGURE 4-6: REGISTER INDIRECT WITH PRE INCREMENT (MODE1, SUBMODE 5)

**4.1.1.7 Mode1, Register Direct with 5-bit Signed Literal**

Addressing MODE1, submode 6/7 is register direct with 5-bit signed literal. As shown in Figure 4-7, operand 1 is contained in  $W_s$ .

Operand 2 is the 5-bit signed literal embedded within the instruction. The 4-bit  $W_b$  field forms the 4 LS-bits of a signed constant. It is concatenated with the LS-bit of the three bit MODE1 field to form the 5-bit signed constant value.

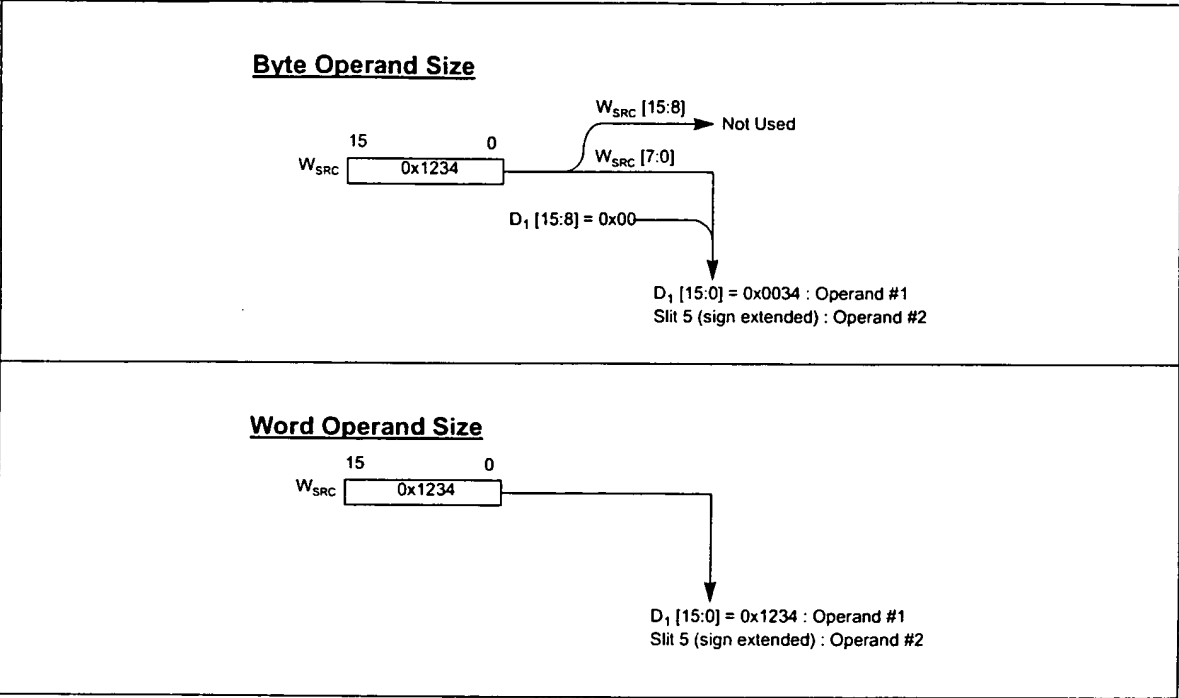


FIGURE 4-7: REGISTER DIRECT WITH 5-BIT SIGNED LITERAL (MODE1, SUBMODE 6/7)

#### 4.1.2 MODE 2

MODE2 determines the addressing mode for either the result destination or a source operand, depending upon instruction requirements. It follows the same definition for each encoding as MODE1 except that it applies to only one operand. The MODE1 signed 5-bit constant value mode makes little sense where MODE2 is used, and is therefore not supported.

In summary, MODE2 supports the addressing mode shown in Table 4-3.

MODE2 Bit Encoding	Function (Source)	Function (Destination)	Description
000	EA = Wsrc	EA = Wdst	Register direct
001	EA = [Wsrc]	EA = [Wdst]	Register indirect
010	EA = [Wsrc]-= 1	EA = [Wdst]-= 1	Register indirect post-decremented
011	EA = [Wsrc]+= 1	EA = [Wdst]+= 1	Register indirect post-incremented
100	EA = [Wsrc-=1]	EA = [Wdst-=1]	Register indirect pre-decremented
101	EA = [Wsrc+=1]	EA = [Wdst+=1]	Register indirect pre-incremented
110	Unused	Unused	
111	Unused	Unused	

TABLE 4-3:MODE 2 ADDRESSING MODE DEFINITION

#### 4.1.2.1 Mode2, Register Direct

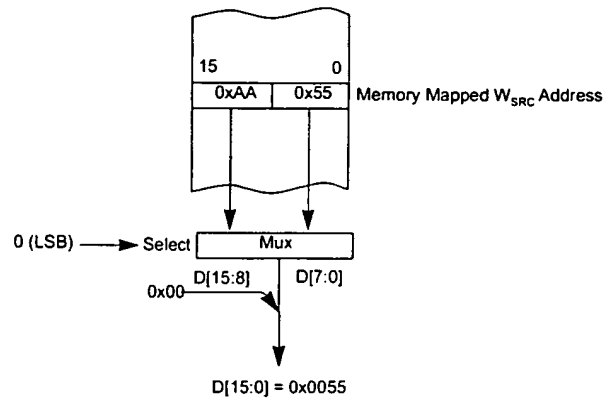
Addressing MODE2, submode 0 is register direct. The implied effective address is the memory mapped address of register Wsrc or Wdst.

The operand is contained in Wsrc as shown in Figure 4-8, or the result is written to Wdst as shown in Figure 4-9. In both cases, Wsrc or Wdst is accessed through addressing its memory mapped image. Note

that, as the EA is implicitly defined as a word address, byte data size accesses will only be able to read or write the LS byte<7:0> (LS-bit of the EA is always clear) in this addressing mode.

**Note:** Rather than executing a memory fetch, it may be preferable to perform two W-array fetches if bussing allows???

##### Byte Operand Size



##### Word Operand Size

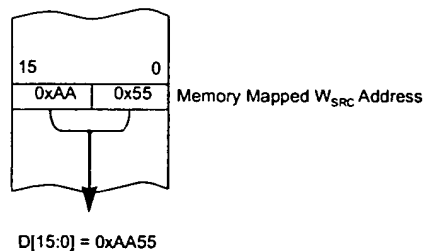


FIGURE 4-8: REGISTER DIRECT, OPERAND SOURCE (MODE2, SUBMODE 0)

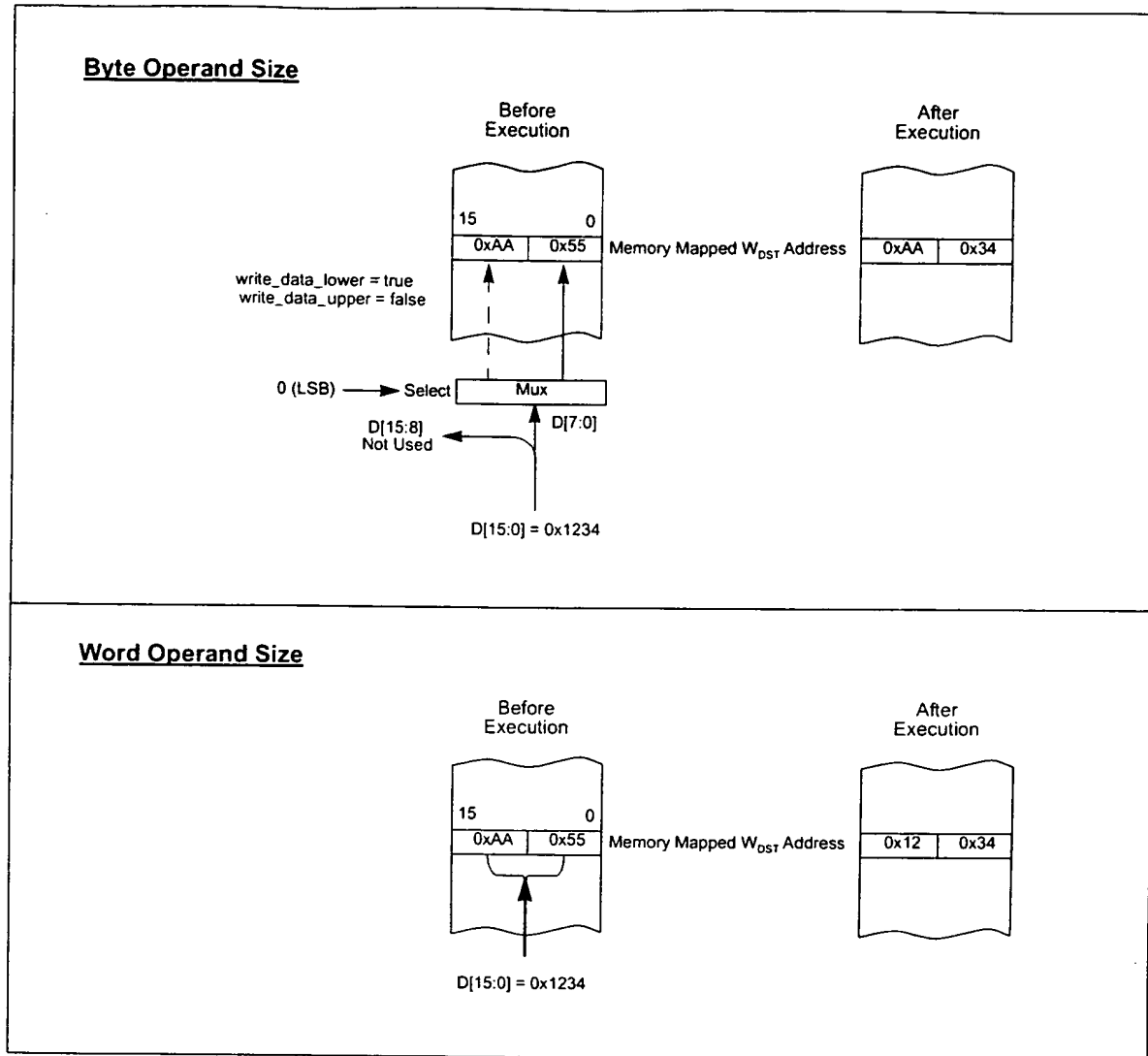


FIGURE 4-9: REGISTER DIRECT, RESULT DESTINATION (MODE2, SUBMODE 0)

#### 4.1.2.2 Mode2, Register Indirect

Addressing MODE2, submode 1 is register indirect. The effective address contained in register Wsrc points to the operand as shown in Figure 4-10, or Wdst points to the result destination as shown in Figure 4-11.

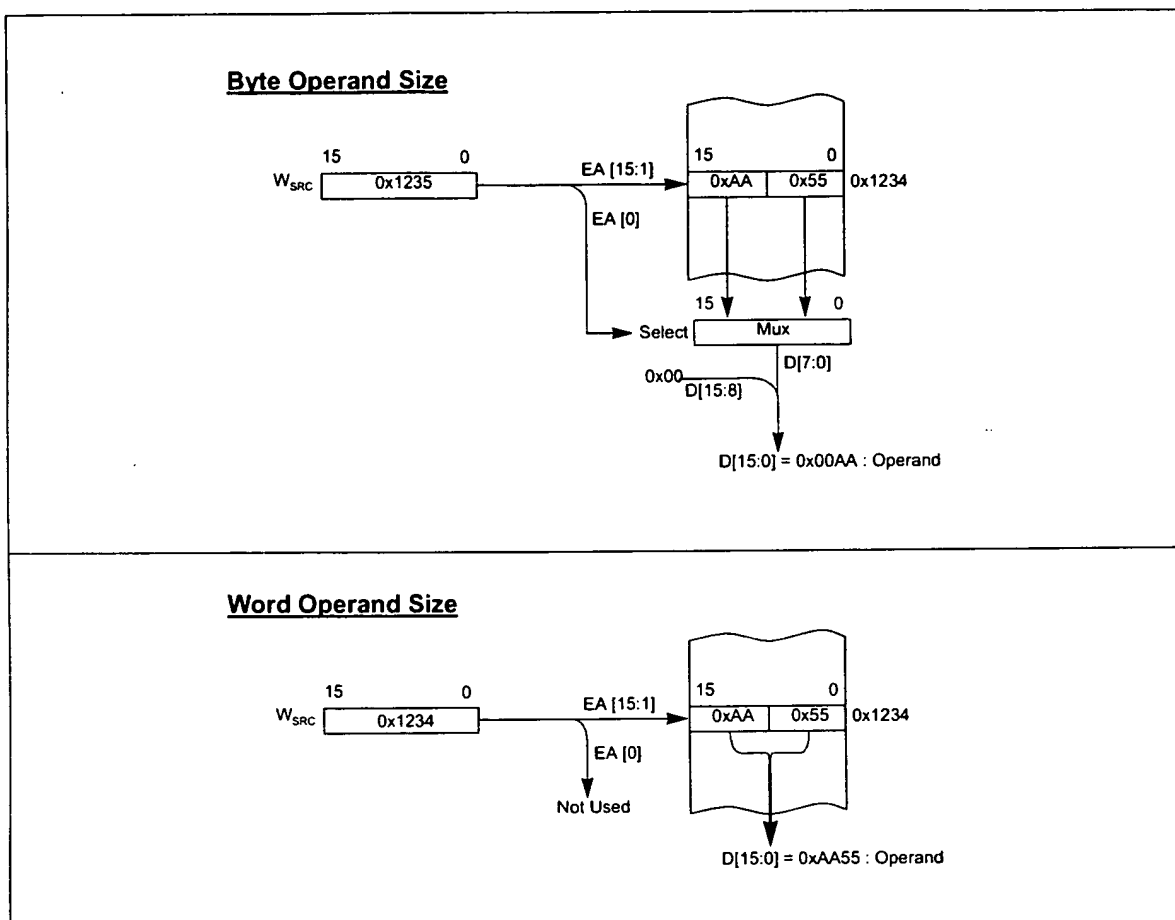


FIGURE 4-10: REGISTER INDIRECT, OPERAND SOURCE (MODE2, SUBMODE 1)



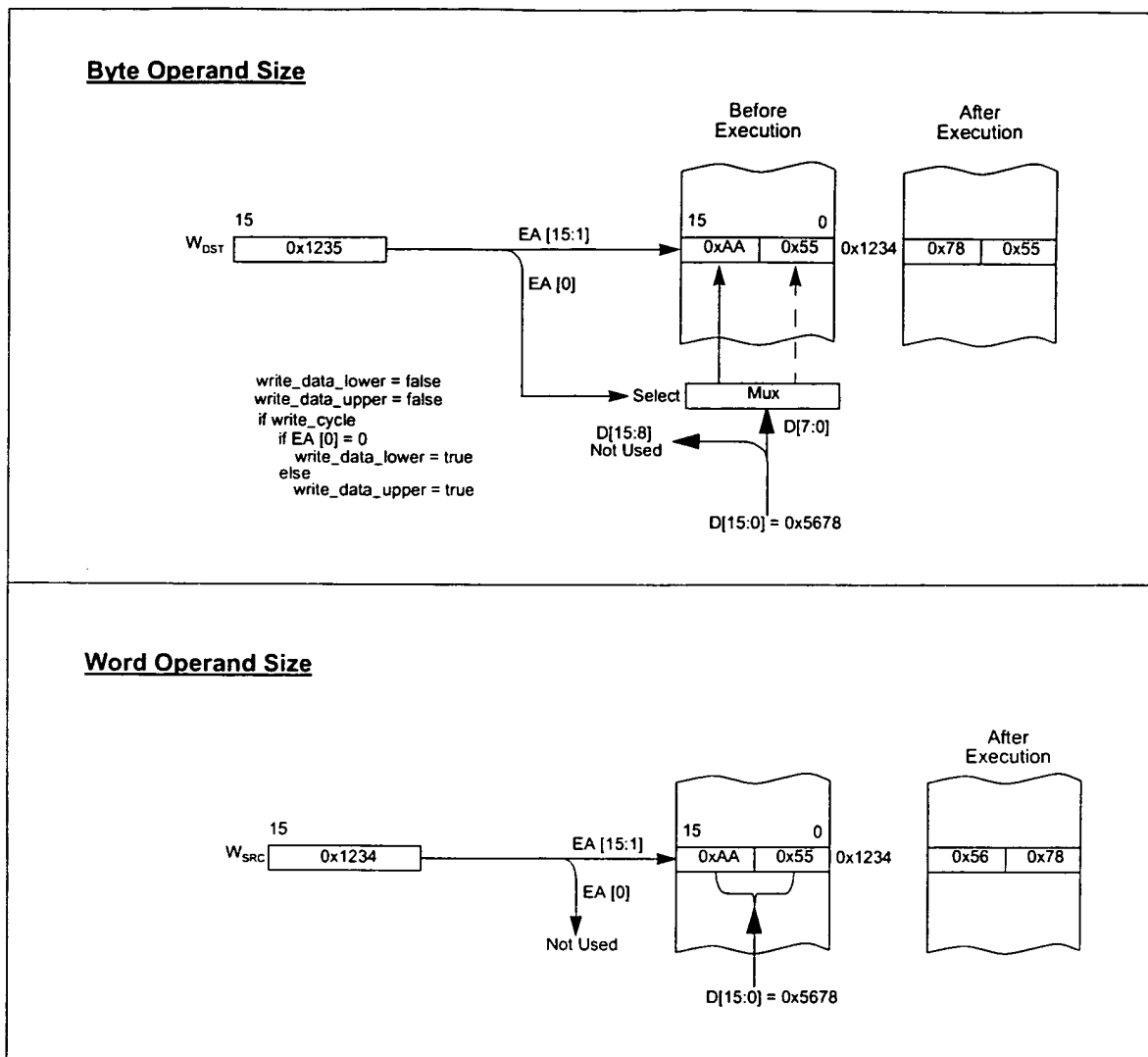


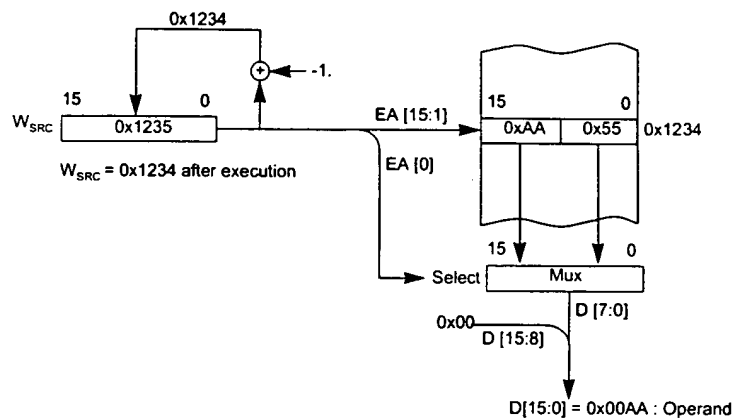
FIGURE 4-11: REGISTER INDIRECT, RESULT DESTINATION (MODE2, SUBMODE 1)

#### 4.1.2.3 Mode2, Register Indirect with Post Decrement

Wsrc or Wdst is then post decremented as shown in Figure 4-12 and Figure 4-13.

Addressing MODE2, submode 2 is register indirect with post decrement. The effective address contained in register Wsrc points to the operand, or the effective address contained in register Wdst points to the result destination.

##### Byte Operand Size



##### Word Operand Size

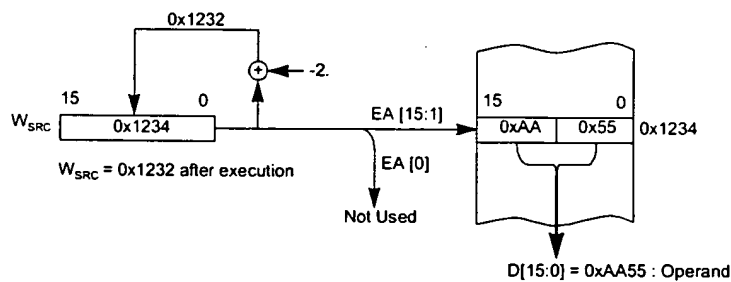


FIGURE 4-12: REGISTER INDIRECT WITH POST DECREMENT, SOURCE OPERAND (MODE2, SUBMODE 2)

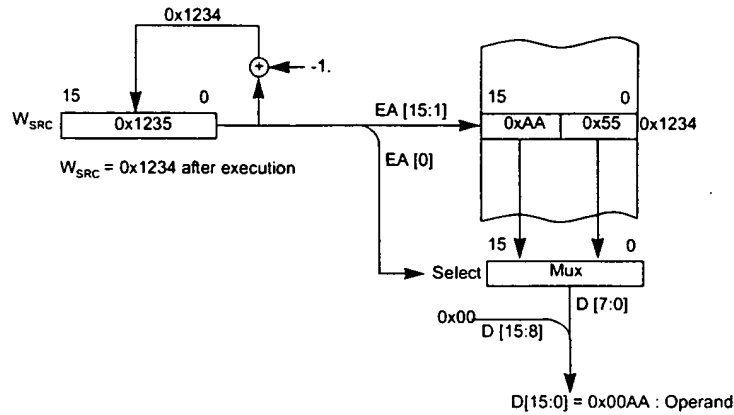
### Byte Operand Size

#### 4.1.2.4 Mode2, Register Indirect with Post Decrement

Addressing MODE2, submode 3 is register indirect with post decrement. The effective address contained in register Wsrc points to the source operand, or the effective address contained in register Wdst points to the result destination

Wsrc or Wdst are then decremented as shown in Figure 4-14 and Figure 4-15.

##### Byte Operand Size



##### Word Operand Size

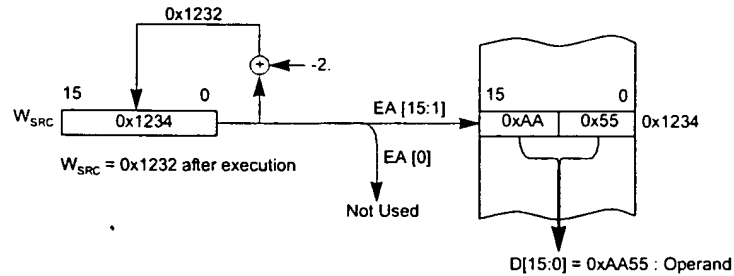


FIGURE 4-14: REGISTER INDIRECT WITH POST INCREMENT, OPERAND SOURCE (MODE2, SUBMODE 3)

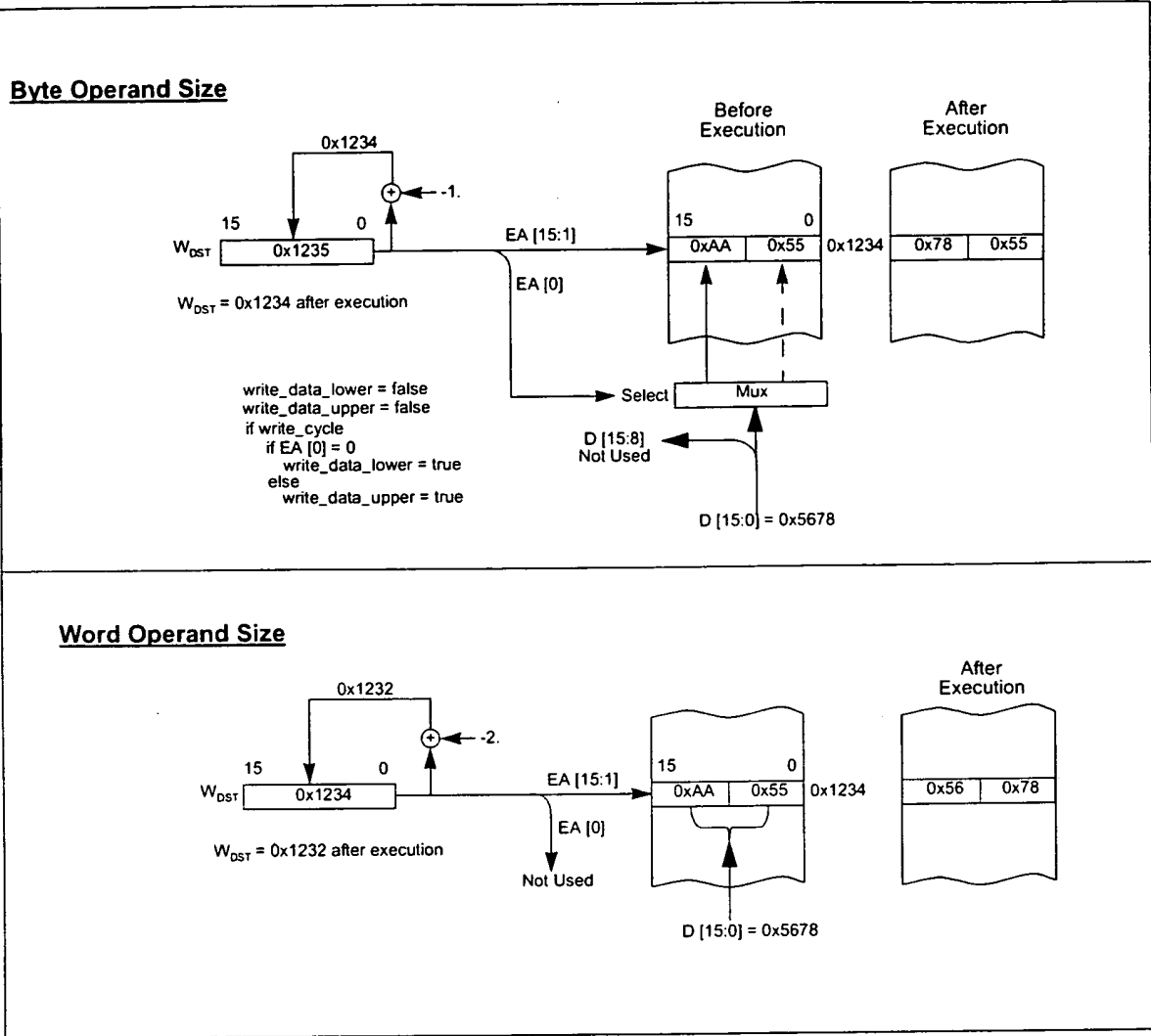


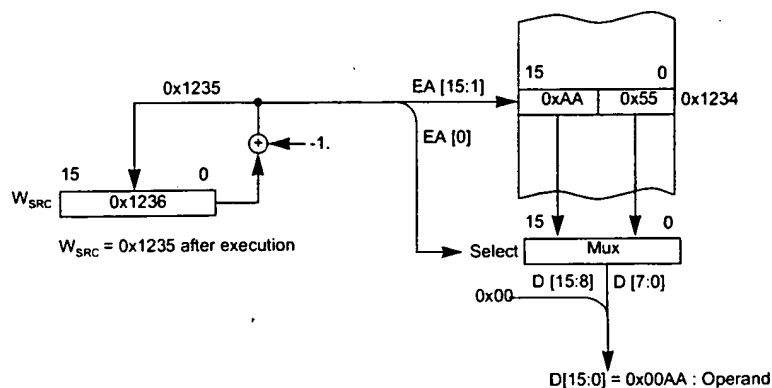
FIGURE 4-15: REGISTER INDIRECT WITH POST INCREMENT, RESULT DESTINATION (MODE2, SUBMODE 3)

#### 4.1.2.5 Mode2, Register Indirect with Pre Decrement

Addressing MODE2, submode 4 is register indirect with pre decrement.

Register Wsrc or Wdst is decremented to form the effective address which points to the operand as shown in Figure 4-18 and Figure 4-19.

##### Byte Operand Size



##### Word Operand Size

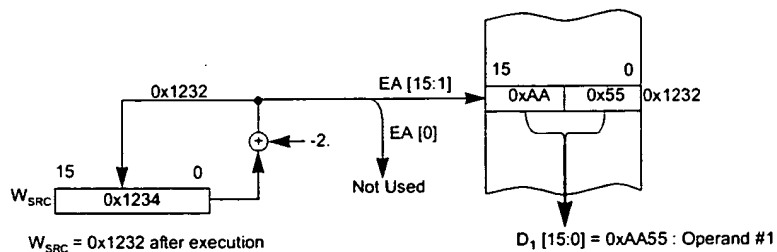
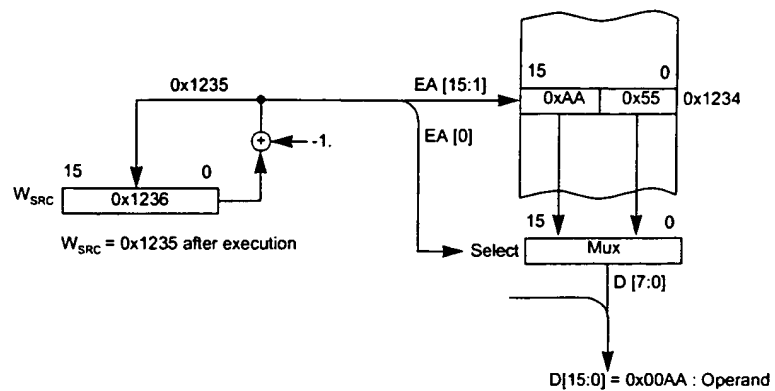


FIGURE 4-16: REGISTER INDIRECT WITH PRE DECREMENT, SOURCE OPERAND (MODE2, SUBMODE 4)

### Byte Operand Size



### Word Operand Size

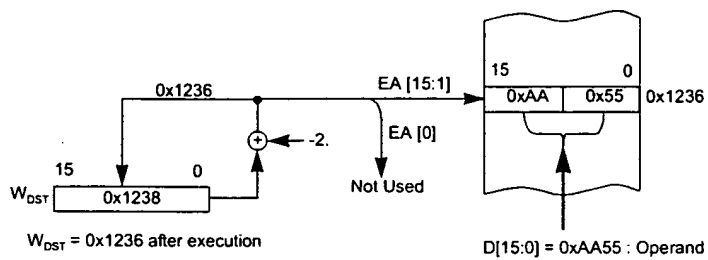


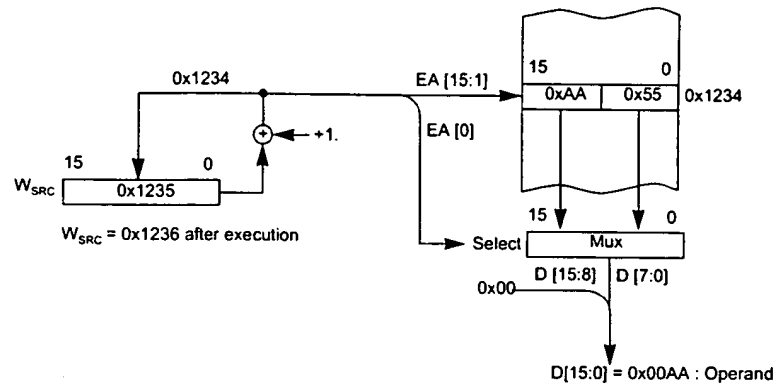
FIGURE 4-17: REGISTER INDIRECT WITH PRE DECREMENT, RESULT DESTINATION (MODE2, SUBMODE 4)

#### 4.1.2.6 Mode2, Register Indirect with Pre Increment

Addressing MODE2, submode 5 is register indirect with pre increment.

Register Wsrc or Wdst is incremented to form the effective address which points to the operand as shown in Figure 4-18 and Figure 4-19.

##### Byte Operand Size



##### Word Operand Size

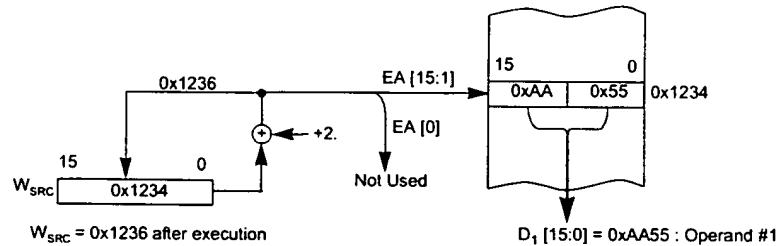
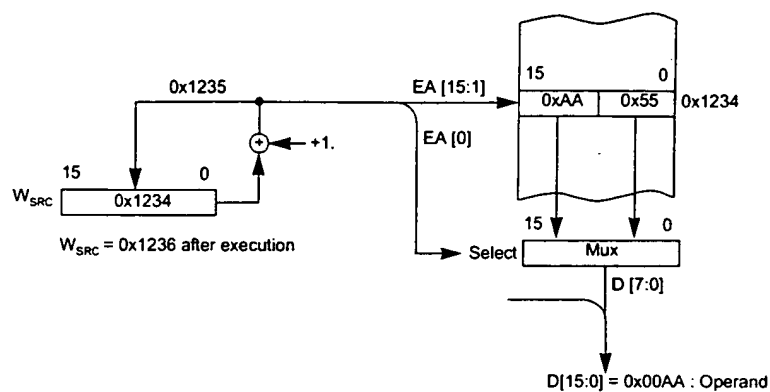


FIGURE 4-18: REGISTER INDIRECT WITH PRE INCREMENT, SOURCE OPERAND (MODE2, SUBMODE 5)



### Byte Operand Size



### Word Operand Size

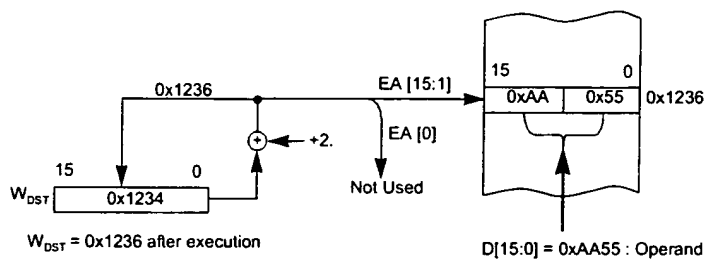


FIGURE 4-19: REGISTER INDIRECT WITH PRE INCREMENT, RESULT DESTINATION (MODE2, SUBMODE 5)

### 4.1.3 MODE 3

MODE3 is used by 'MOVE' and some of the DSP class instructions where addressing flexibility is important. It follows the same definition for each encoding as MODE1 except that it uses the Wb field as an address operand (instead of a data operand). In addition, MODE3 also supports register with register offset addressing mode, sometimes referred to as register indexed.

The 5-bit signed constant required by submode 6/7 is created by concatenating the Wb field with the LS-bit of the 3-bit MODE3 field.

**Note:** For the MOV instruction, the MODE3 addressing modes can differ for the source and destination EA. However, the 4-bit Wb field is shared between both source and destination (but typically only used by one).

In summary, MODE3 supports the addressing mode shown in Table 4-4

MODE3 Bit Encoding	Function	Description
000	EA = Wn	Register direct
001	EA = [Wn]	Register indirect
010	EA = [Wdst]-= 1	Register indirect post-decremented
011	EA = [Wdst]+= 1	Register indirect post-incremented
100	EA = [Wdst- 1]	Register indirect pre-decrement
101	EA = [Wn + Wb]	Register indirect with register offset
110	EA = [Wn + S5lit]	Register indirect with signed 5-bit
111		constant value offset (note 1)

TABLE 4-4:MODE 3 ADDRESSING MODE DEFINITION

09870457-060101

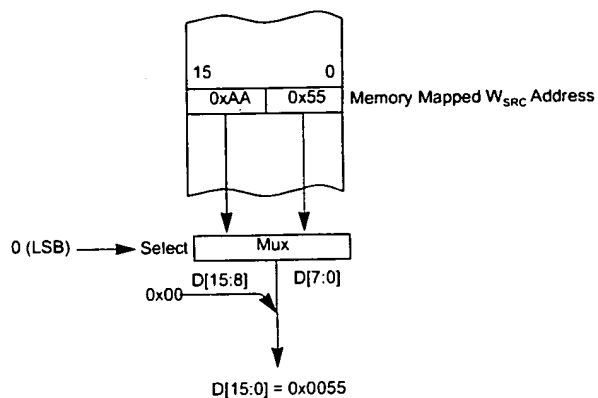
#### 4.1.3.1 Mode3, Register Direct

Addressing MODE3, submode 0 is register direct. The implied effective address is the memory mapped address of register Wsrc or Wdst.

The operand is contained in Wsrc as shown in Figure 4-20, or the result is written to Wdst as shown in Figure 4-22. In both cases, Wsrc or Wdst is accessed through addressing its memory mapped image.

**Note:** Rather than executing a memory fetch, it may be preferable to perform two W-array fetches if bussing allows???

##### Byte Operand Size



##### Word Operand Size

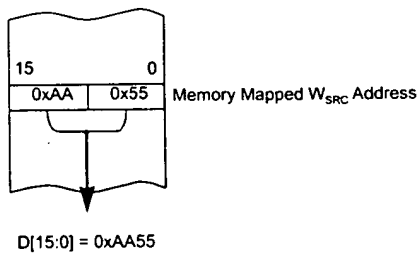


FIGURE 4-20: REGISTER DIRECT, OPERAND SOURCE (MODE3, SUBMODE 0)

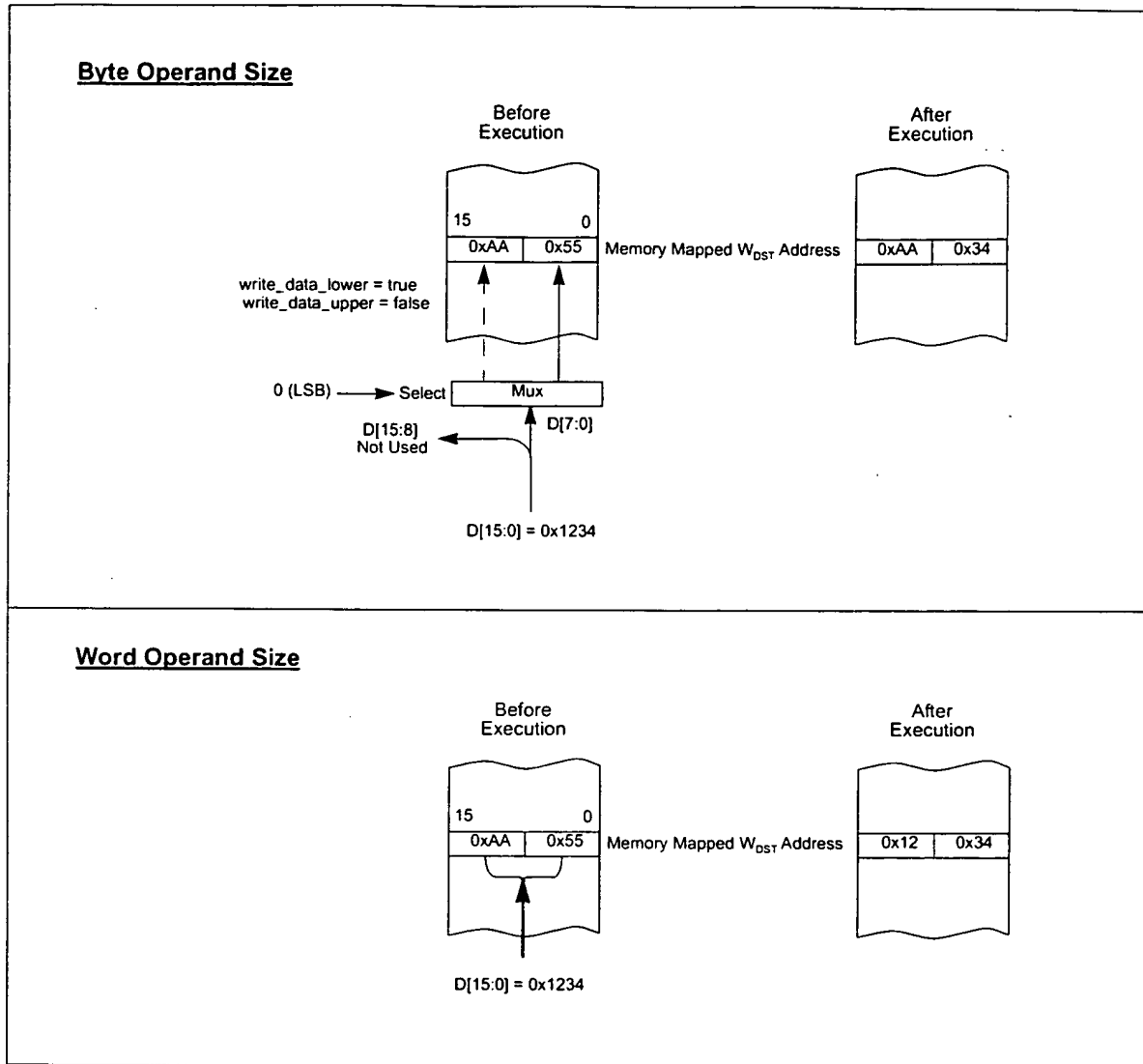


FIGURE 4-21: REGISTER DIRECT, OPERAND SOURCE (MODE3, SUBMODE 0)

**SECRET**

1



TOP SECRET 45402360

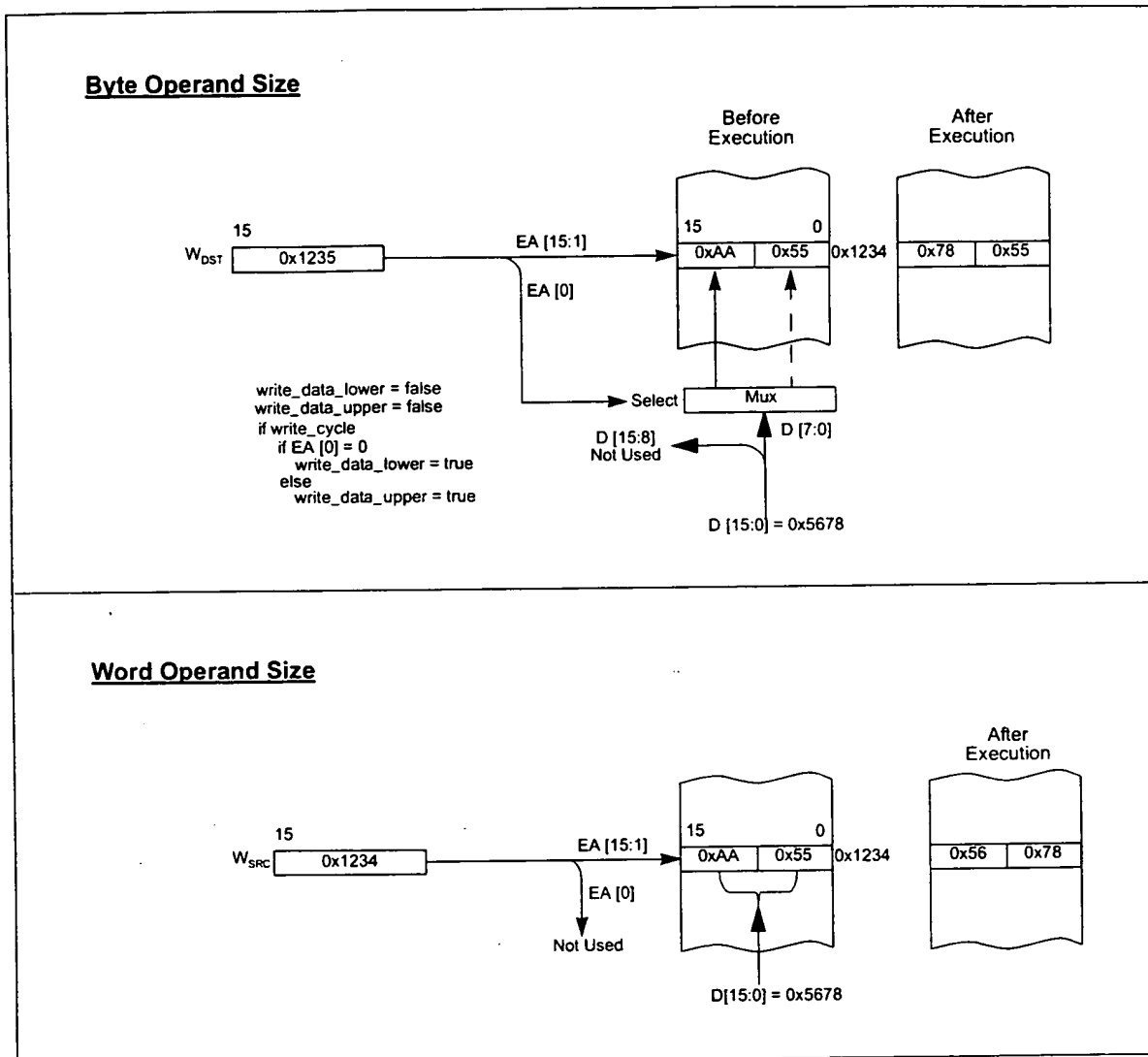


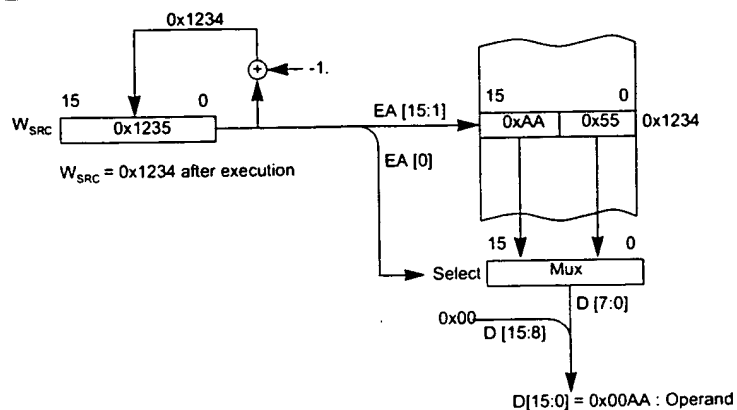
FIGURE 4-23: REGISTER INDIRECT, RESULT DESTINATION (MODE3, SUBMODE 1)

#### 4.1.3.3 Mode3, Register Indirect with Post Decrement

Addressing MODE3, submode 2 is register indirect with post decrement. The effective address contained in register Wsrc points to the operand, or the effective address contained in register Wdst points to the result destination.

Wsrc or Wdst is then post decremented as shown in Figure 4-24 and Figure 4-28.

##### Byte Operand Size



##### Word Operand Size

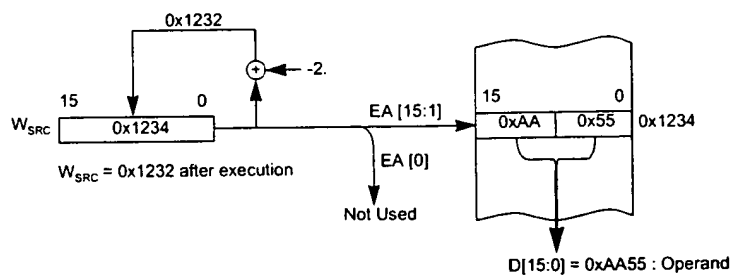


FIGURE 4-24: REGISTER INDIRECT WITH POST DECREMENT, SOURCE OPERAND (MODE3, SUBMODE 2)

TOP SECRET 45402360

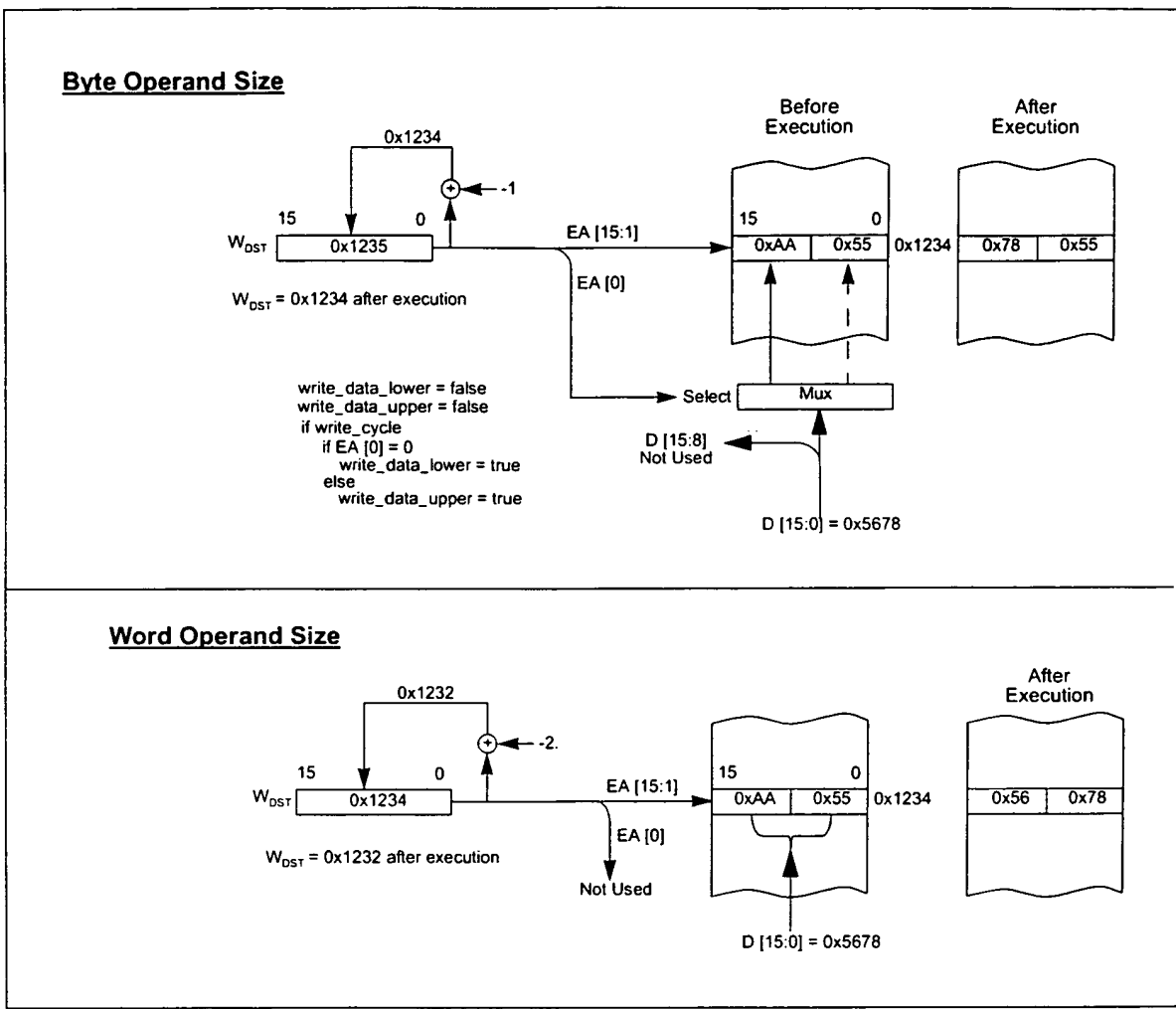


FIGURE 4-25: REGISTER INDIRECT WITH POST DECREMENT, RESULT DESTINATION (MODE3, SUBMODE 2)



#### 4.1.3.4 Mode3, Register Indirect with Post Modification

Wsrc or Wdst are then incremented as shown in Figure 4-26 and Figure 4-27.

Addressing MODE3, submode 3 is register indirect with post-increment. The effective address contained in register Wsrc points to the operand or the effective address contained in register Wdst points to the result destination.

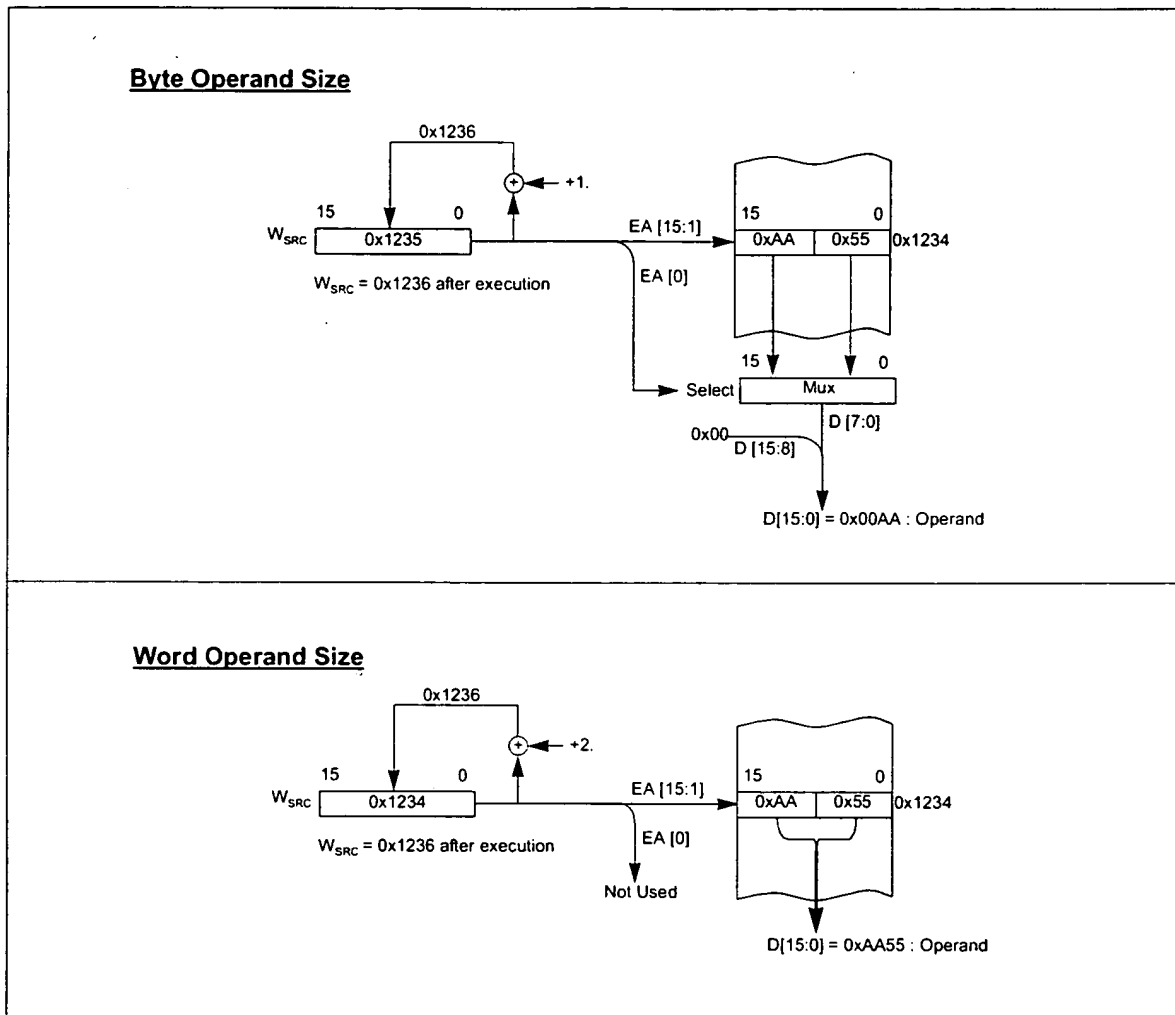


FIGURE 4-26: REGISTER INDIRECT WITH POST INCREMENT, SOURCE OPERAND (MODE3, SUBMODE 3)

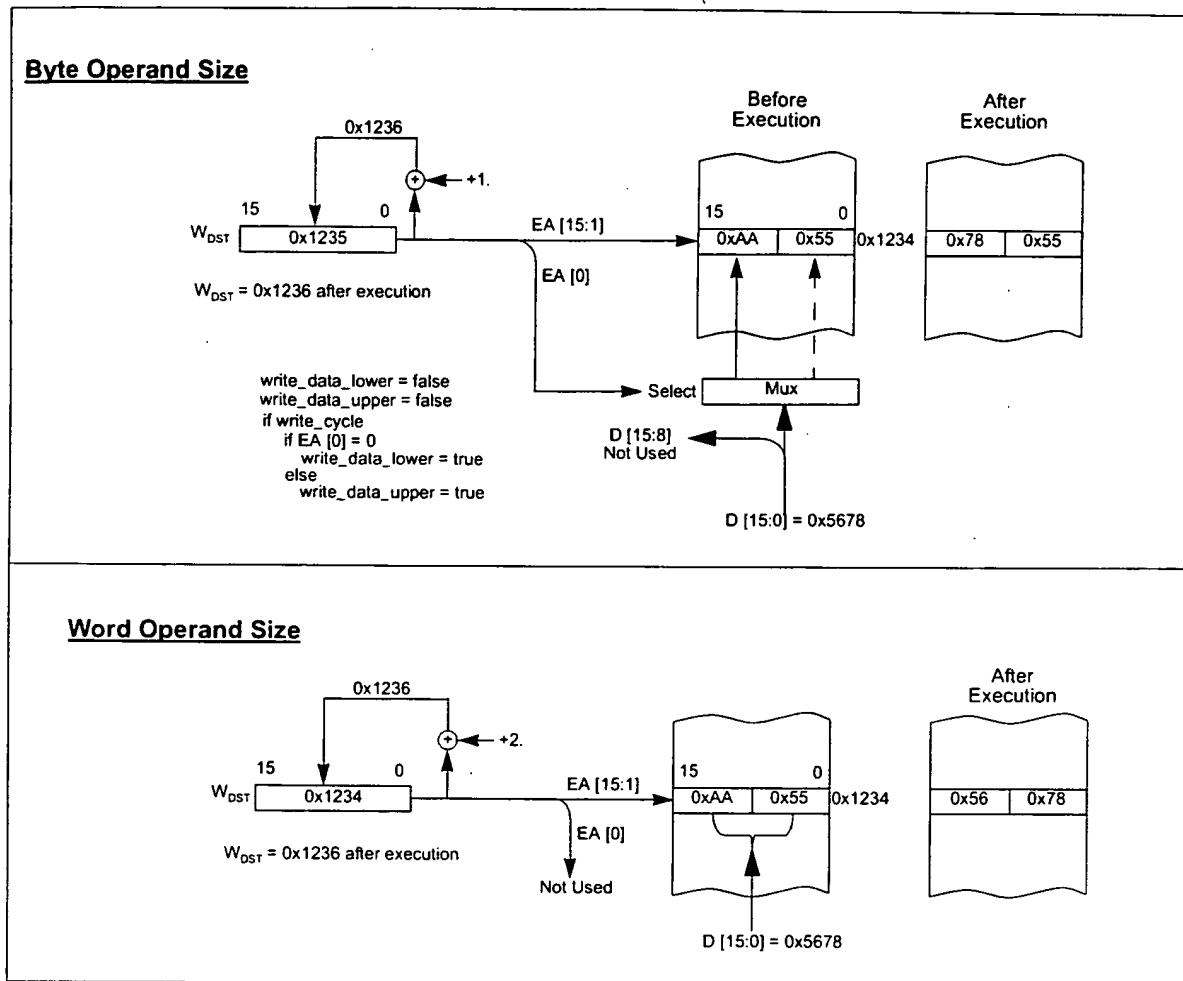


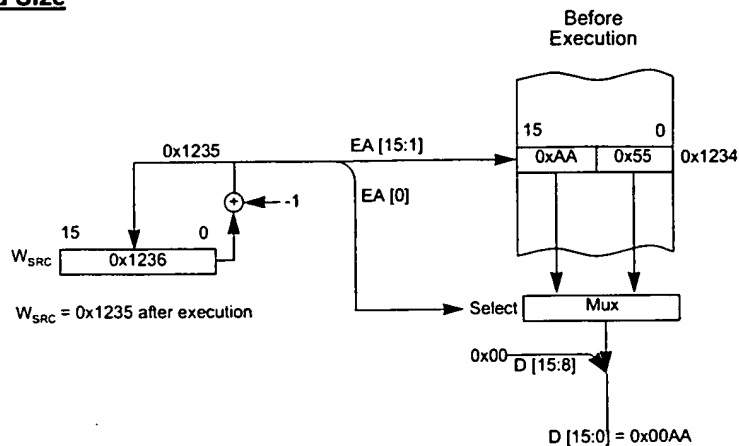
FIGURE 4-27: REGISTER INDIRECT WITH POST INCREMENT, RESULT DESTINATION (MODE3, SUBMODE 3)

#### 4.1.3.5 Mode3, Register Indirect with Pre Decrement

Addressing MODE2, submode 4 is register indirect with pre decrement.

Register Wsrc or Wdst is decremented to form the effective address which points to the operand as shown in Figure 4-28 and Figure 4-29..

##### Byte Operand Size



##### Word Operand Size

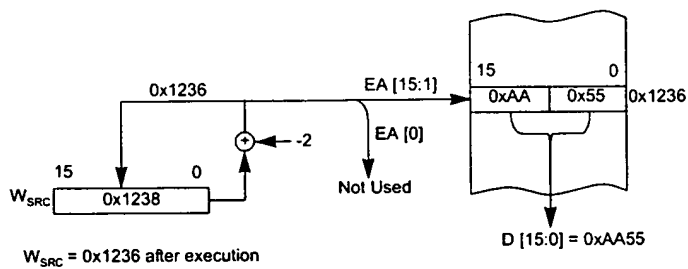
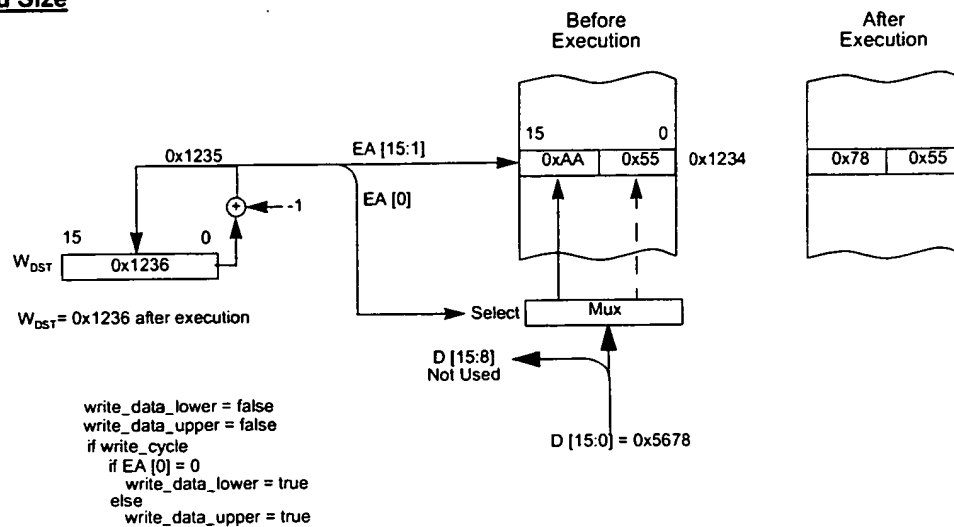


FIGURE 4-28: REGISTER INDIRECT WITH PRE DECREMENT, SOURCE OPERAND (MODE3, SUBMODE 4)

### Byte Operand Size



### Word Operand Size

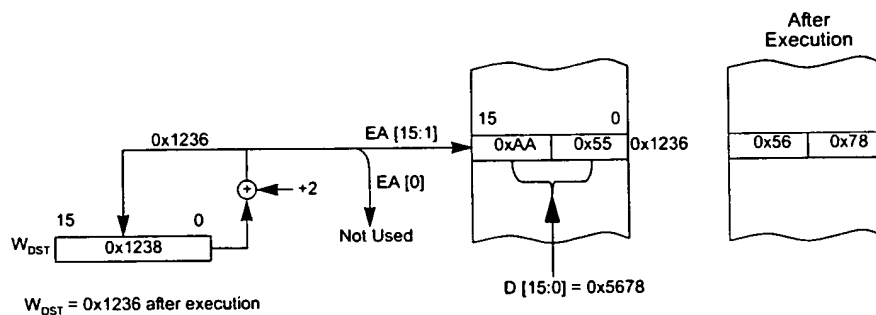


FIGURE 4-29: REGISTER INDIRECT WITH PRE DECREMENT, RESULT DESTINATION (MODE3, SUBMODE 4)



TOP SECRET 060457 250296

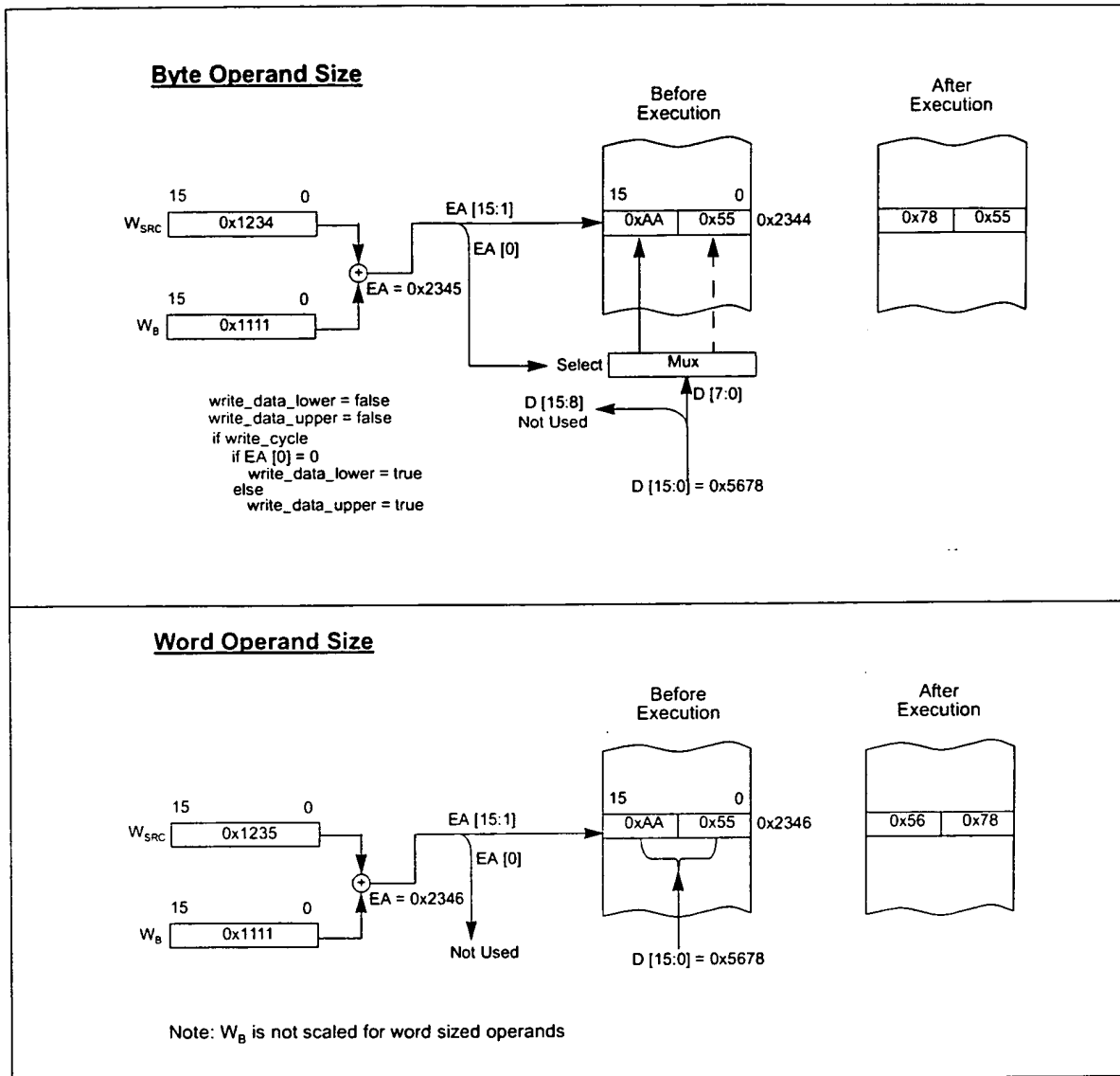


FIGURE 4-31: REGISTER INDIRECT WITH REGISTER OFFSET, RESULT DESTINATION (MODE3, SUBMODE 5)

#### 4.1.3.7 Mode3, Register Indirect with Constant Offset

Addressing MODE3, submode 6/7 is register indirect with constant offset. For an operand read, the effective address of the operand is formed by adding the contents of Wsrc and a 5-bit signed literal, as shown in Figure 4-32. For a result destination write, the effective address of the operand is formed by adding the con-

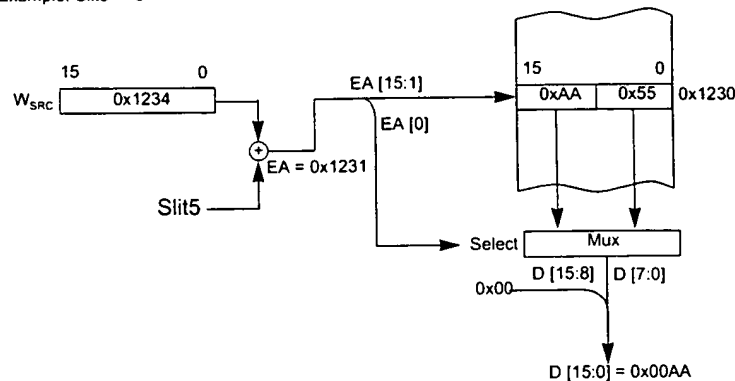
tents of Wdst and a 5-bit signed literal as shown in Figure 4-33. Wsrc or Wdst are not modified by these operations.

The 4-bit Wb field forms the 4 LS-bits of the signed constant. It is concatenated with the LS-bit of the three bit MODE1 field to form a 5-bit signed constant value.

If the 5-bit signed literal equals 0, this addressing mode is interpreted as register indirect with a pre-decrement..

##### Byte Operand Size

Example: Slit5 = -3



##### Word Operand Size

Example: Slit5 = -1

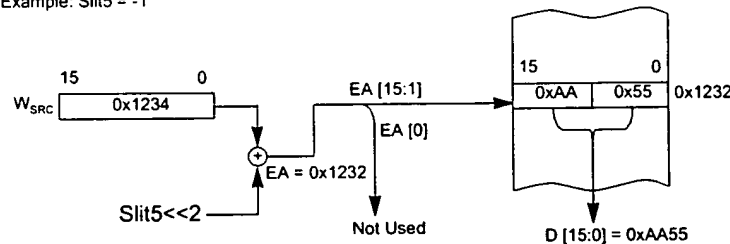


FIGURE 4-32: REGISTER INDIRECT WITH CONSTANT OFFSET (I=0), SOURCE OPERAND (MODE3, SUBMODE 6/7)

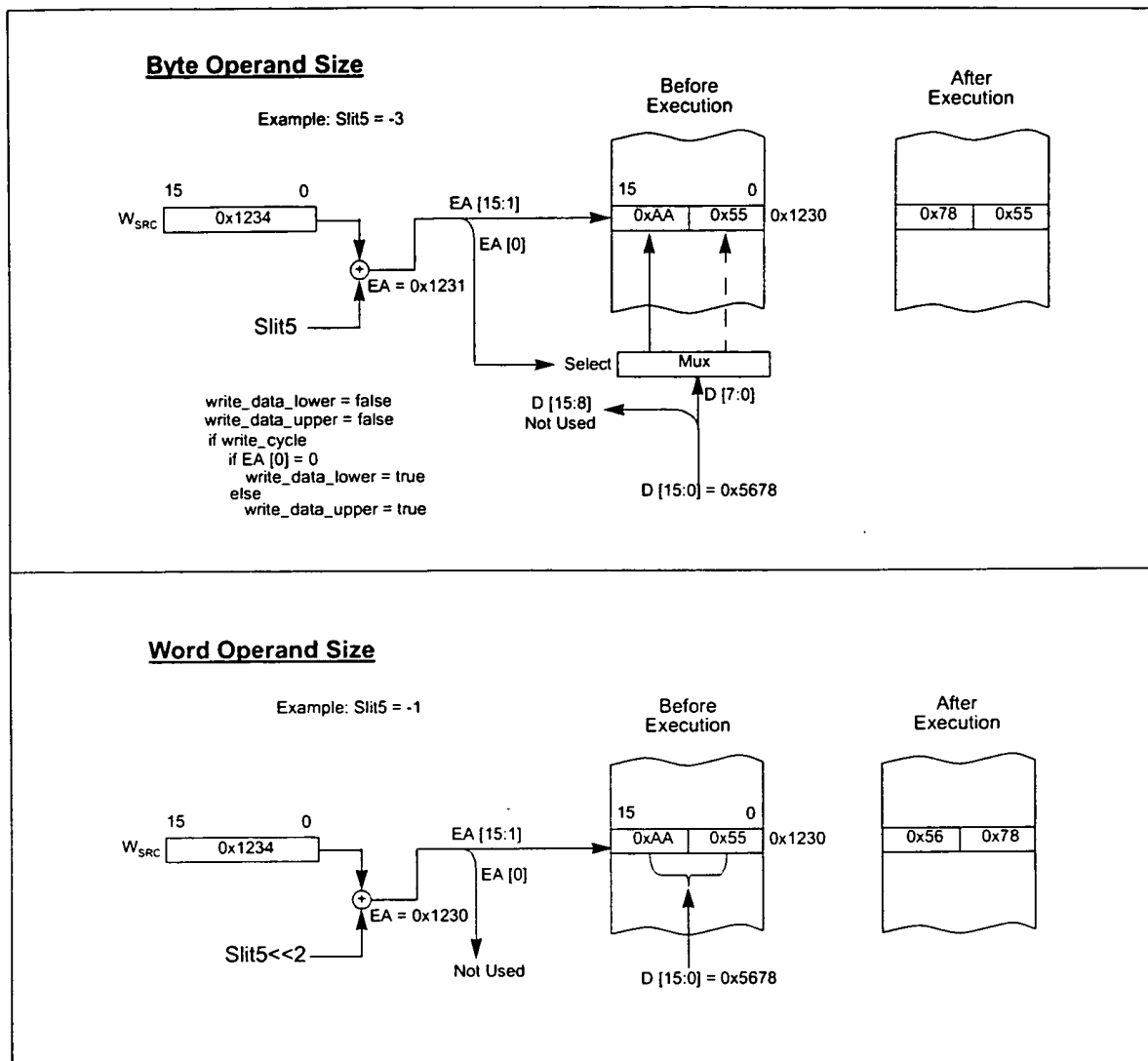
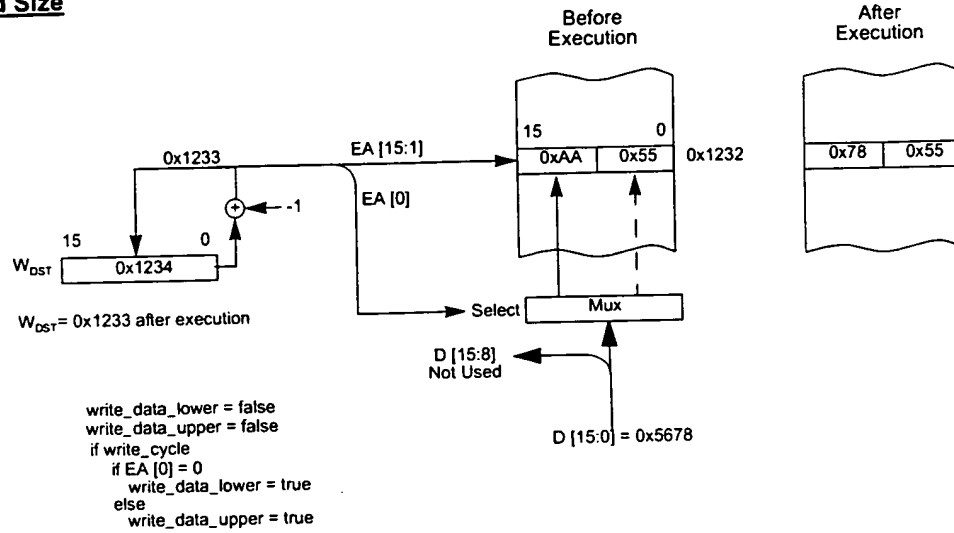


FIGURE 4-33: REGISTER INDIRECT WITH CONSTANT OFFSET (I=0), RESULT DESTINATION (MODE3, SUBMODE 6/7)



### Byte Operand Size

### Byte Operand Size



### Word Operand Size

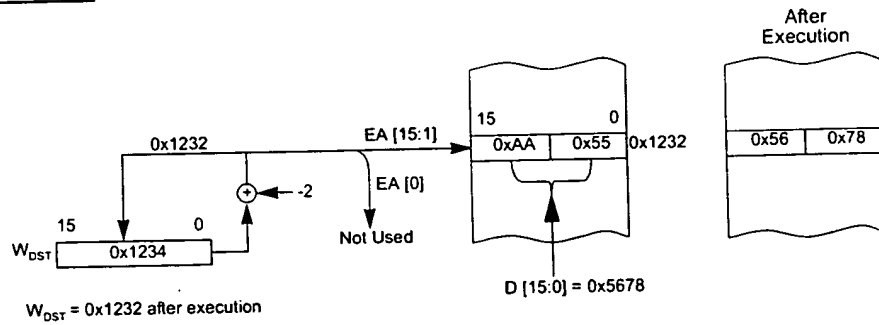


FIGURE 4-35: REGISTER INDIRECT WITH PRE DECREMENT, RESULT DESTINATION (MODE3, SUBMODE 6 WITH LITERAL OFFSET = 0)

#### 4.1.4 MODE 4

The dual source operand DSP instructions (MAC, CLRAC, MPYAC & MOVAC) utilize a simplified set of addressing modes (MODE4) to allow the user to effectively manipulate the data pointers through register indirect tables.

Wsrc must be a member of the set {W4, W5, W6, W7}. For data reads, W4 and W5 will always be directed to the X AGU and W6 and W7 will always be directed to the Y AGU. The effective addresses generated (before

and after modification) must therefore be valid addresses within X data space for W4 and W5, and Y data space for W6 and W7.

**Note:** Register indirect with register offset addressing is only available for W5 (in X space) and W7 (in Y space).

In summary, MODE3 supports the addressing modes shown in Table 4-5 for X data space and those shown in Table 4-6 for Y data space.

MODE4 Bit Encoding	Function	Description
0000	EA = [W <sub>4</sub> ]	Register indirect
0001	EA = [W <sub>4</sub> ]+2	Register indirect post-inc by 2
0010	EA = [W <sub>4</sub> ]+4	Register indirect post-inc by 4
0011	EA = [W <sub>4</sub> ]+6	Register indirect post-inc by 6
0100	None	Disable data pre-fetch
0101	EA = [W <sub>4</sub> ]-6	Register indirect post-dec by 6
0110	EA = [W <sub>4</sub> ]-4	Register indirect post-dec by 4
0111	EA = [W <sub>4</sub> ]-2	Register indirect post-dec by 2
1000	EA = [W <sub>5</sub> ]	Register indirect
1001	EA = [W <sub>5</sub> ]+2	Register indirect post-inc by 2
1010	EA = [W <sub>5</sub> ]+4	Register indirect post-inc by 4
1011	EA = [W <sub>5</sub> ]+6	Register indirect post-inc by 6
1100	EA = [W <sub>5</sub> +W <sub>8</sub> ]	Register indirect with register offset (indexed)
1101	EA = [W <sub>5</sub> ]-6	Register indirect post-dec by 6
1110	EA = [W <sub>5</sub> ]-4	Register indirect post-dec by 4
1111	EA = [W <sub>5</sub> ]-2	Register indirect post-dec by 2

- Note 1: MODE4 instructions are word sized only, so post-modification values are already scaled appropriately  
2: Addressing mode defined by read address space

TABLE 4-5:MODE 4 ADDRESSING MODE DEFINITION FOR X DATA SPACE

MODE4 Bit Encoding	Function	Description
0000	$EA = [W_6]$	Register indirect
0001	$EA = [W_6] + 2$	Register indirect post-inc by 2
0010	$EA = [W_6] + 4$	Register indirect post-inc by 4
0011	$EA = [W_6] + 6$	Register indirect post-inc by 6
0100	None	Disable data pre-fetch
0101	$EA = [W_6] - 6$	Register indirect post-dec by 6
0110	$EA = [W_6] - 4$	Register indirect post-dec by 4
0111	$EA = [W_6] - 2$	Register indirect post-dec by 2
1000	$EA = [W_7]$	Register indirect
1001	$EA = [W_7] + 2$	Register indirect post-inc by 2
1010	$EA = [W_7] + 4$	Register indirect post-inc by 4
1011	$EA = [W_7] + 6$	Register indirect post-inc by 6
1100	$EA = [W_7 + W_8]$	Register indirect with register offset
1101	$EA = [W_7] - 6$	Register indirect post-dec by 6
1110	$EA = [W_7] - 4$	Register indirect post-dec by 4
1111	$EA = [W_7] - 2$	Register indirect post-dec by 2

Note 1: MODE4 instructions are word sized only, so post-modification values are already scaled appropriately  
 2: Addressing mode defined by read address space

**TABLE 4-6: MODE 4 ADDRESSING MODE DEFINITION FOR Y DATA SPACE**

#### 4.1.4.1 Mode4, Register Indirect

Addressing MODE4, submodes 0 & 8 are register indirect. The effective address contained in register Wsrc points to the operand as shown in Figure 4-36. Only word sized operands are allowed.

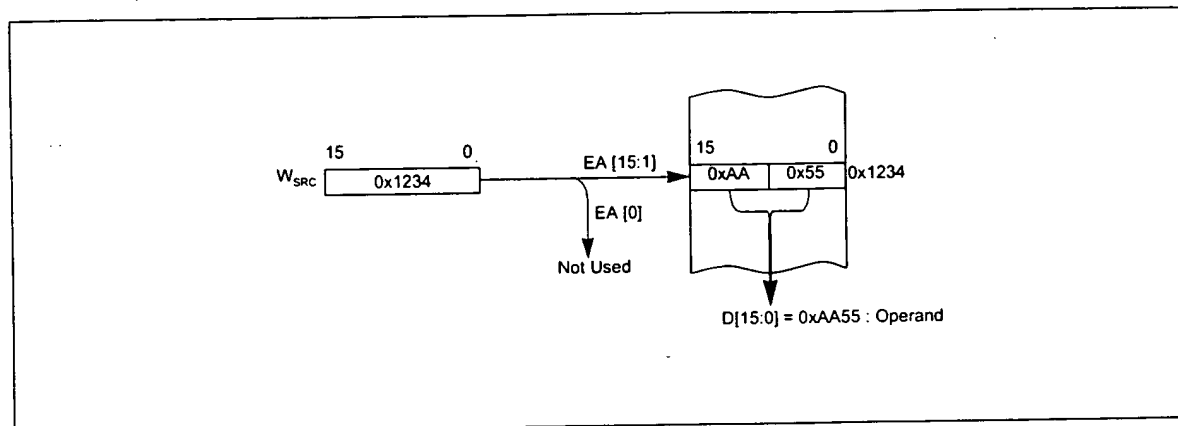


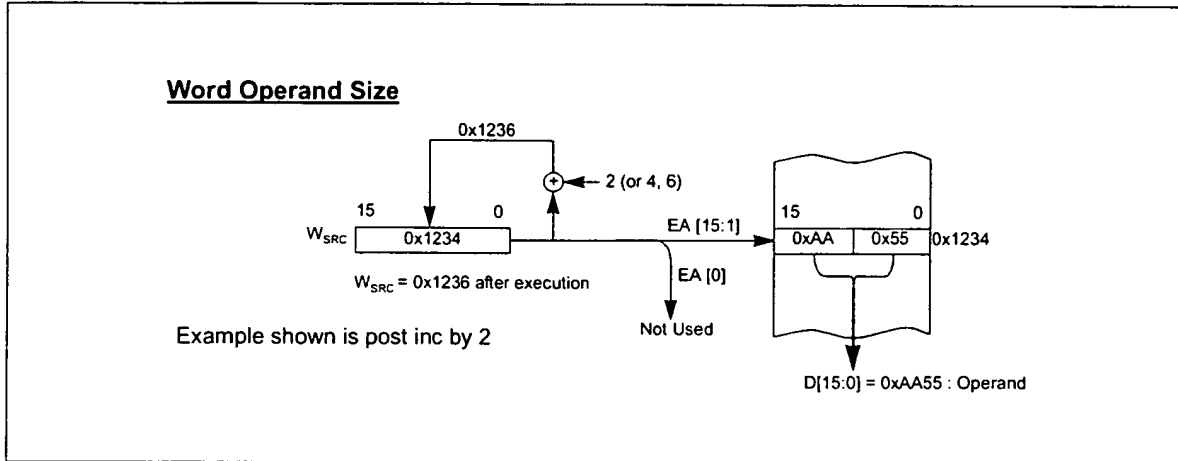
FIGURE 4-36: REGISTER INDIRECT (MODE4, SUBMODE 0 AND 8)

#### 4.1.4.2 Mode4, Register Indirect with Post Increment

Addressing MODE4, submodes 1, 2, 3, 9, 10 & 11 are register indirect with post increment. The effective address contained in register Wsrc points to the operand.

Wsrc is then post incremented by 2, 4 or 6 as shown in Figure 4-37.

**Note:** Misaligned word fetches are possible if Wsrc contains an odd value. Should this occur, an address error trap will be generated.



**FIGURE 4-37: REGISTER INDIRECT WITH POST INCREMENT (MODE4, SUBMODES 1, 2, 3, 9, 10 & 11)**

---

#### 4.1.4.3 Mode4, Pre-fetch Inhibit

Addressing mode MODE4, submode 4 will inhibit a data fetch from X or Y address space. No target registers are modified (the target register selection is a don't care).

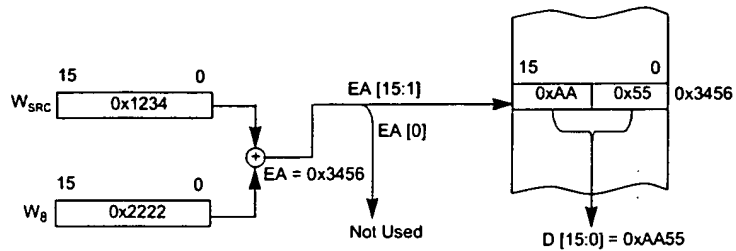
09870457-060101

#### 4.1.4.4 Mode4, Register Indirect with Register Offset

Addressing MODE4, submodes 12 is register indirect with register offset. The effective address of the operand is formed by adding the contents of Wsrc (W5 or W7) and W8 as shown in Figure 4-30. The offset register is fixed as W8. Neither Wsrc or W8 are not modified by these operations.

**Note:** This addressing mode operates in an identical manner to that of Mode3 register indirect with register offset, in which the offset register (W8 in this case) is not automatically scaled for word accesses. Consequently, misaligned word fetches are possible if W8 contains an odd value. Should this occur, an address error trap will be generated.

##### Word Operand Size



Note: W<sub>8</sub> is not scaled for word sized operands

FIGURE 4-38: REGISTER INDIRECT WITH REGISTER OFFSET, OPERAND SOURCE (MODE4, SUBMODE 12)



#### 4.1.4.5 Mode4, Register Indirect with Post Decrement

Addressing MODE4, submodes 5, 6, 7, 13, 14 & 15 are register indirect with post decrement. The effective address contained in register Wsrc points to the operand.

Wsrc is then post decremented by 2, 4 or 6 as shown in Figure 4-39.

**Note:** Misaligned word fetches are possible if Wsrc contains an odd value. Should this occur, an address error trap will be generated.

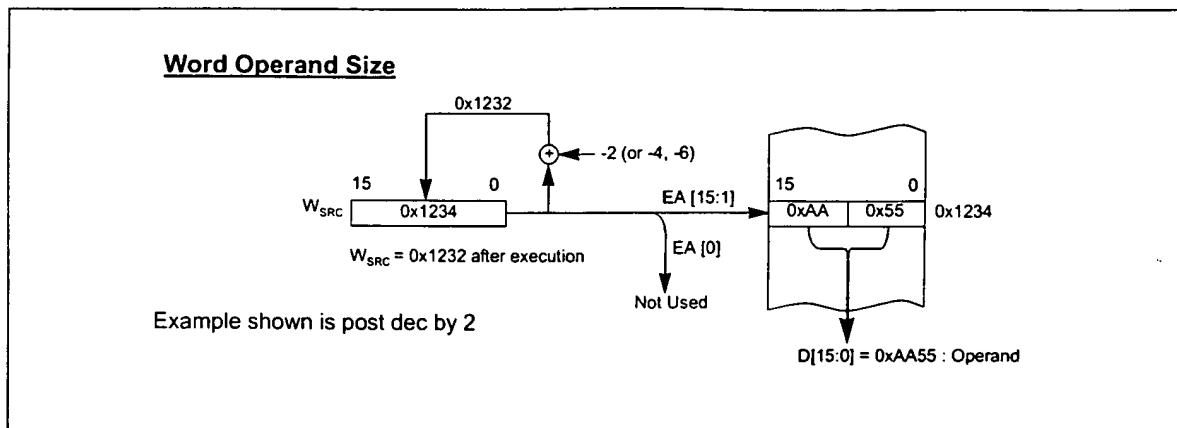


FIGURE 4-39: REGISTER INDIRECT WITH POST DECREMENT (MODE4, SUBMODES 5, 6, 7, 13, 14 & 15)

## 4.2 X AGU

See Section 4.4.1 for more details and examples.

The X AGU supports all addressing modes including modulo addressing and bit reversed addressing. A block diagram is shown in Figure 4-40. The basic elements are now described.

### 4.2.1 Effective Address Adder

The effective address (EA) adder generates the effective addresses for all instruction using X data space prior to modification by modulo addressing. It supports all addressing modes including bit reversed addressing. The adder accepts the source or destination W register on the A input and either of the following on B input based upon which addressing mode is required.

1. Offset (Wb) register contents
2. Signed 5-bit literal, Slit5
3. Constant value of:  
0, +1, +2, +4, +6, -1, -2, -4 or -6

The value range for Slit5 is  $-16 \leq \text{Slit5} \leq +15$ .

### 4.2.2 Modulo and Bit Reversed Addressing Controller

The Modulo and Bit Reversed Addressing Controller block enables or disables these addressing modes, and provides the appropriate control signals to the rest of the AGU. If modulo and bit reversed addressing are disabled, the EA adder result passes unmodified to the AGU output. See Section 4.5 for more details.

### 4.2.3 Modulo Addressing Comparator/Subtractor

Modulo addressing relies on automatic correction of any generated EA such that it is forced back into the selected circular buffer address range.

For an incrementing buffer, the offset sign is positive. The end address is therefore routed to the subtractor, and subtracted from the new EA. If the result is negative, the address is within the buffer boundaries and will propagate unchanged. If the result is positive (including zero), indicating the EA has passed the end address, it is logically ORed with the start address. This is equivalent to adding it to the start address to create the wrap address for a start address on a 'zero' power of two boundary.

For a decrementing buffer, the offset sign is negative. The start address is therefore routed to the subtractor, and subtracted from the new EA. If the result is positive, the address is within the buffer boundaries and will propagate unchanged. If the result is negative, indicating the EA has passed the start address, it is logically ANDed with the start address. This is equivalent to adding it (a negative value) to the start address to create the wrap address for an end address on a 'ones' address boundary.



## 4.3 Y AGU

As the Y AGU is only used by the MAC class of DSP instructions, its function is restricted to supporting post-modified register indirect (using a constant modifier) and modulo addressing. A block diagram is shown in Figure 4-41. The basic elements are now described.

### 4.3.1 Effective Address Adder

The effective address (EA) Adder generates the effective addresses for all instruction using Y data space prior to modification by modulo addressing. It supports post-modified register indirect (using a constant modifier). It does not support bit reversed addressing. The adder accepts the source or destination W register on the A input and a constant (0, +2, +4, +6, -2, -4 or -6) on B input, depending upon the post modified constant declared in the instruction.

### 4.3.2 Modulo Addressing Controller

The Modulo Addressing Controller block enables or disables modulo addressing, and provides the appropriate control signals to the rest of the AGU. If modulo addressing is disabled, the EA adder result passes unmodified to the AGU output. See Section 4.5 for more details.

### 4.3.3 Modulo Addressing Comparator/Subtractor

Modulo addressing relies on automatic correction of any generated EA such that it is forced back into the selected circular buffer address range.

For an incrementing buffer, the offset sign is positive. The end address is therefore routed to the subtractor, and subtracted from the new EA. If the result is negative, the address is within the buffer boundaries and will propagate unchanged. If the result is positive (including zero), indicating the EA has passed the end address, it is logically ORed with the start address. This is equivalent to adding it to the start address to create the wrap address for a start address on a 'zero' power of two boundary.

For a decrementing buffer, the offset sign is negative. The start address is therefore routed to the subtractor, and subtracted from the new EA. If the result is positive, the address is within the buffer boundaries and will propagate unchanged. If the result is negative, indicating the EA has passed the start address, it is logically ANDed with the start address. This is equivalent to adding it (a negative value) to the start address to create the wrap address for an end address on a 'ones' address boundary.

See Section 4.4.1 for more details and examples.



## 4.4 Modulo Addressing

Modulo addressing is a method of providing an automated means to support circular data buffers using hardware. The objective is to remove the need for software to perform data address boundary checks when executing tightly looped code as is typical in many DSP algorithms.

dsPIC modulo addressing can operate in either data or program space (since the data pointer mechanism is essentially the same for both). One circular buffer can be supported in each of the X (which also provides the pointers into Program space) and Y data spaces. Modulo addressing can operate on any W register pointer.

In order to minimize the hardware size for modulo addressing support, certain usage restrictions are imposed which are discussed in detail Section 4.4.1 and Section 4.4.6. In summary, any one circular buffer can only be allowed to operate in one direction as the buffer start address (for incrementing buffers) or end address (for decrementing buffers) is restricted based upon the direction of the buffer. The direction is determined from the address offset sign.

### 4.4.1 Start and End Address

The modulo addressing scheme requires that either a starting or an end address be specified and loaded into the 16-bit modulo buffer address registers, XMODSRT, XMODEND, YMODSRT, YMODEND.

The data buffer start address is arbitrary but must be at a 'zero', power of two boundary for incrementing address buffers. It can be any address for decrementing address buffers. For example, if the buffer size (modulus value) is chosen to be 100 bytes (0x64), then the buffer start address for an incrementing buffer must contain 7 least significant zeros. Valid start addresses may therefore be 0xXX00 and 0xXX80 where 'x' is any hexadecimal value. Adding the buffer length to this value will give the end address to be written into X/YMODEND. For example, if the start address was chosen to be 0x2000, then the X/YMODEND would be set to  $(0x2000 + 0x0064) = 0x2064$ . Note that the last physical address of the buffer will be at end address -1 because the buffer range is 0 to 0x63.

**Note:** 'Starting address' refers to the smallest address boundary of the circular buffer. The initial entry address (first access of the buffer) may point to any address within the modulus range (see Section 4.4.6).

The data buffer end address is arbitrary but must be at a 'ones' boundary for decrementing buffers. It can be at any address for an incrementing buffer. For example, if the buffer size (modulus value) is chosen to be 100 bytes (0x64), then the buffer end address for an incrementing buffer must contain 7 least significant ones. Valid end addresses may therefore be 0xFFFF

and 0xXX7F where 'X' is any hexadecimal value. Subtracting the buffer length from this value then adding 1 will give the start address to be written into X/YMODSRT. For example, if the end address was chosen to be 0x207F, then the start address would be  $(0x207F - 0x0064 + 1) = 0x201C$ , which is the first physical address of the buffer.

In an incrementing buffer, the modulo addressing hardware performs the address correction by subtracting the buffer end address from the EA and, if the result is positive, adding it to the start address. As the start address is on a 'zero', power of two boundary, the addition may be performed by a logical OR operation.

In an decrementing buffer, the modulo addressing hardware performs the address correction by subtracting the buffer start address from the EA and, if the result is negative, adding it to the end address. As the end address is on a 'ones' boundary, the addition may be performed by a logical AND operation.

**Note:** All modulo addressing EA calculations assume word size data (LS-bit of every EA is always clear). The XM value is scaled accordingly to generate compatible (byte) addresses, leaving the LS-bit of all EAs clear.

### 4.4.2 Buffer Length

The data buffer length can be any value up to 64K words. The buffer length is not used in this scheme to correct buffer addresses or determine modulo range.

### 4.4.3 W Address Register Selection

The modulo and bit reversed addressing control register MODCON<15:0> contains enable flags plus W register field to specify the W address registers. The XWM and YWM fields select which registers will operate with modulo addressing. If XWM = 15, AGU X modulo addressing is disabled. Similarly, if YWM = 15, AGU Y modulo addressing is disabled.

Modulo addressing and bit reversed addressing should not be enabled together. In the event that the user attempts to do this, bit reversed addressing will assume priority when active and X modulo addressing will be disabled.

The X address space pointer W register (XWM) to which modulo addressing is to be applied, is stored in MODCON<3:0> (see Register 4-1). Modulo addressing is enabled for X data space when XWM is set to any value other than 15 and the XMODEN bit is set at MODCON[15].

The Y address space pointer W register (YWM) to which modulo addressing is to be applied, is stored in MODCON<7:4> (see Register 4-2). Modulo addressing is enabled for Y data space when YWM is set to any value other than 15 and the YMODEN bit is set at MODCON[14].

#### 4.4.4 Modulo Addressing Applicability

Modulo addressing can be applied to the effective address (EA) calculation associated with any W register. It is important to realize that the address boundaries checks look for addresses less than or greater than the upper (for incrementing buffers) and lower

(for decrementing buffers) boundary addresses (not just equal to). Address changes may therefore jump over boundaries and still be adjusted correctly (see Section 4.4.6 for restrictions).

#### 4.4.5 Modulo Addressing Operation

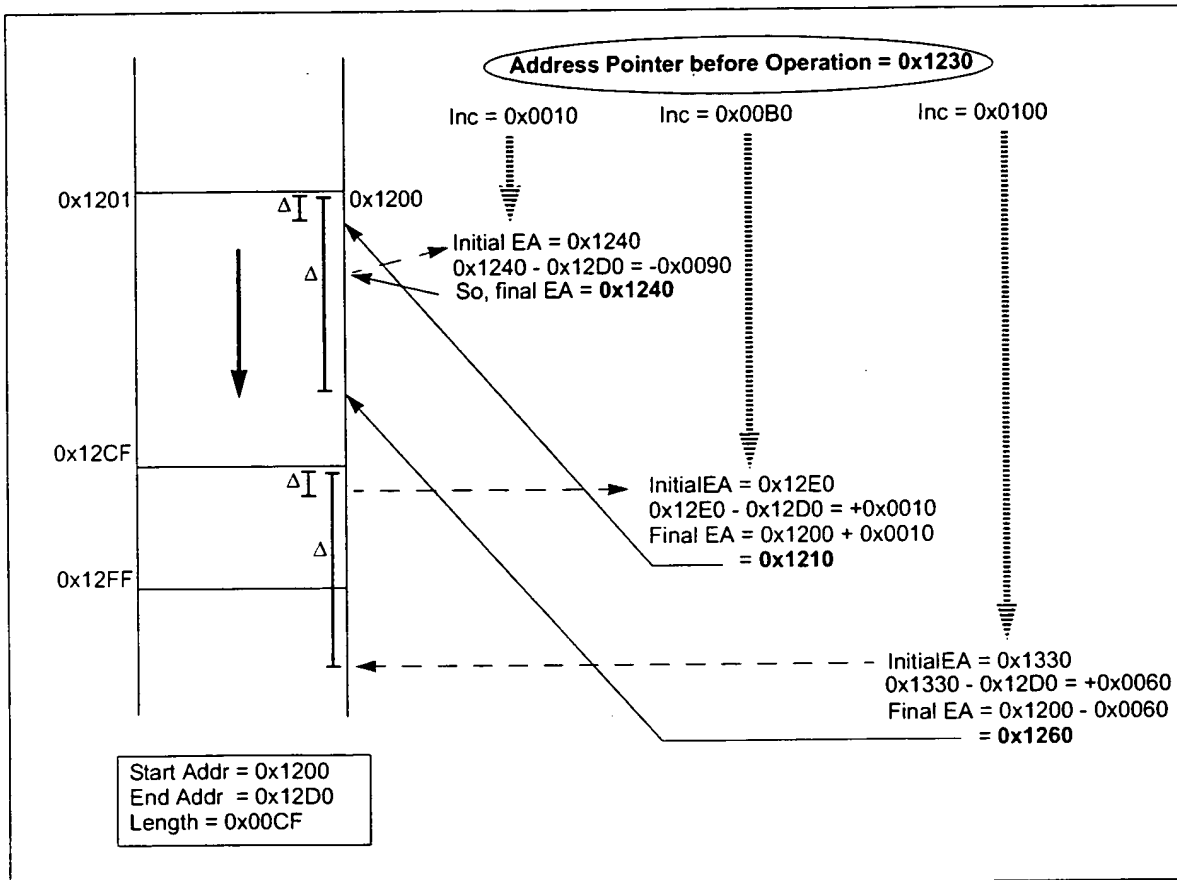


FIGURE 4-42: INCREMENTING BUFFER MODULO ADDRESSING OPERATION EXAMPLE

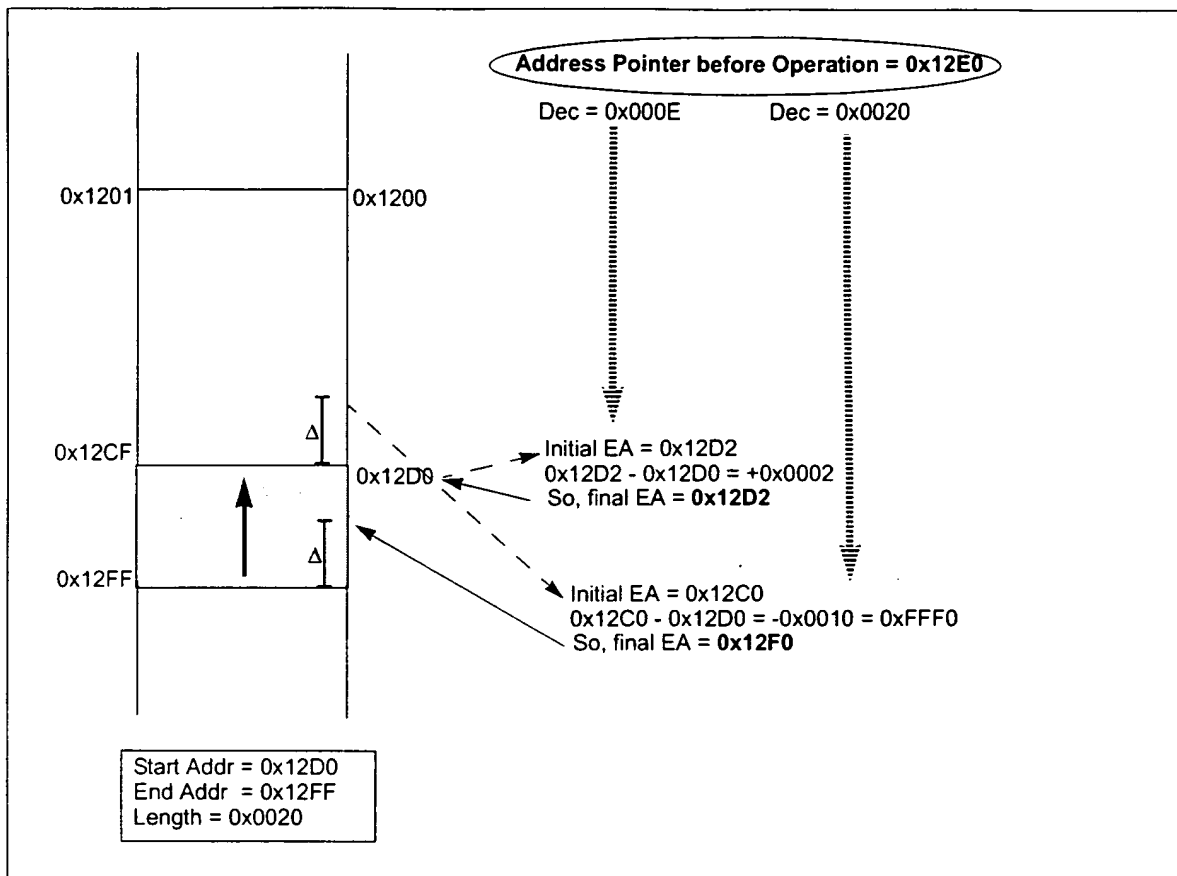


FIGURE 4-43: DECREMENTING BUFFER MODULO ADDRESSING OPERATION EXAMPLE



# **REGISTER 4-1: MODCON, MODULO & BIT REVERSED ADDRESSING CONTROL REGISTER (0XXXXX)**

Upper Half:							
R/W-0	R/W-0	U	U	R/W-0	R/W-0	R/W-0	R/W-0
XMODEN	YMODEN	-	-	BWM3	BWM2	BWM1	BWM0
bit 15				bit 8			

Lower Half:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
YWM3	YWM2	YWM1	YWM0	XWM3	XWM2	XWM1	XWM0
bit 7				bit 0			

- bit 15 **XMODEN:** X AGU Modulus Addressing Enable  
1 = X AGU Modulus Addressing enabled  
0 = X AGU Modulus Addressing disabled
- bit 14 **YMODEN:** Y AGU Modulus Addressing Enable  
1 = Y AGU Modulus Addressing enabled  
0 = Y AGU Modulus Addressing disabled
- bit 13 Unused
- bit 12 Unused
- bit 11-8 **BWM:** X AGU Register Select for Bit Reversed Addressing  
0000 = W0 selected for bit reversed addressing  
| |  
1110 = W14 selected for bit reversed addressing  
  
1111 = W15 bit reversed addressing disabled
- bit 7-4 **YWM:** Y AGU W Register Select for Modulo Addressing  
0000 = W0 selected for modulo addressing  
| |  
1110 = W14 selected for modulo addressing  
  
1111 = W15 modulo addressing disabled
- bit 3-0 **XWM:** X AGU W Register Select for Modulo Addressing  
0000 = W0 selected for modulo addressing  
| |  
1110 = W14 selected for modulo addressing  
  
1111 = W15 modulo addressing disabled

Legend			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	1 = bit is set	0 = bit is cleared	x = bit is unknown

**REGISTER 4-2: XMODSRT, X AGU MODULO ADDRESSING START REGISTER (XXXXh)**

Upper Half:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
XS15	XS14	XS13	XS12	XS11	XS10	XS9	XS8
bit 15 <span style="float: right;">bit 8</span>							

Lower Half:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
XS7	XS6	XS5	XS4	XS3	XS2	XS1	XS0
bit 7 <span style="float: right;">bit 0</span>							

bit 15-0 XS: X AGU Modulo Addressing Start Address

**Legend**

R = Readable bit  
-n = Value at POR

W = Writable bit  
1 = bit is set

U = Unimplemented bit, read as '0'  
0 = bit is cleared  
x = bit is unknown

**REGISTER 4-3: XMODEND, X AGU MODULO ADDRESSING END REGISTER (XXXXh)**

Upper Half:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
XE15	XE14	XE13	XE12	XE11	XE10	XE9	XE8
bit 15 <span style="float: right;">bit 8</span>							

Lower Half:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
XE7	XE6	XE5	XE4	XE3	XE2	XE1	XE0
bit 7 <span style="float: right;">bit 0</span>							

bit 15-0 XE: X AGU Modulo Addressing End Address

**Legend**

R = Readable bit  
-n = Value at POR

W = Writable bit  
1 = bit is set

U = Unimplemented bit, read as '0'  
0 = bit is cleared  
x = bit is unknown

#### REGISTER 4-4: YMODSRT, Y AGU MODULO ADDRESSING START REGISTER (XXXXh)

Upper Half:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
YS15	YS14	YS13	YS12	YS11	YS10	YS9	YS8
bit 15 <span style="float: right;">bit 8</span>							

Lower Half:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
YS7	YS6	YS5	YS4	YS3	YS2	YS1	YS0
bit 7 <span style="float: right;">bit 0</span>							

bit 15-0 YS: Y AGU Modulo Addressing Start Address

##### Legend

R = Readable bit  
-n = Value at POR

W = Writable bit  
1 = bit is set

U = Unimplemented bit, read as '0'  
0 = bit is cleared  
x = bit is unknown

#### REGISTER 4-5: YMODEND, Y AGU MODULO ADDRESSING END REGISTER (XXXXh)

Upper Half:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
YE15	YE14	YE13	YE12	YE11	YE10	YE9	YE8
bit 15 <span style="float: right;">bit 8</span>							

Lower Half:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
YE7	YE6	YE5	YE4	YE3	YE2	YE1	YE0
bit 7 <span style="float: right;">bit 0</span>							

bit 15-0 YE: X AGU Modulo Addressing End Address

##### Legend

R = Readable bit  
-n = Value at POR

W = Writable bit  
1 = bit is set

U = Unimplemented bit, read as '0'  
0 = bit is cleared  
x = bit is unknown

#### 4.4.6 Modulo Addressing Restrictions

As stated in Section 4.4.1, for an incrementing buffer the circular buffer start address (lower boundary) is arbitrary but must be at a 'zero', power of two boundary. For a decrementing buffer, the circular buffer end address is arbitrary but must be at a 'ones' boundary.

With this scheme, there are no restriction regarding how much an EA calculation can exceeds the address boundary being checked, and still be successfully corrected.

Once configured, the direction of successive addresses into a buffer cannot be changed. Although all EA's will continue to be generated correctly irrespective of offset sign, only one address boundary is checked for each type of buffer. Accessing an incrementing buffer with a decrementing address could result in the address decrementing through the start address. If this occurs, an out of range address will be detected but the address wrap operation will fail unless the end address is on a 'ones' address boundary (because the addition is simplified to an OR operation). For example, if the start address = 0x2000, end addresses that will support a bi-directional buffer include 0x200F, 0x203F or any modulo 2 length buffer.

As similar augment applies to accessing a decrementing buffer with an incrementing address.

#### 4.4.7 Modulo Addressing Timing

Modulo addressing can operate on both source and destination operands (i.e. for data reads and writes). Consequently, it must meet timing for the standard instruction cycle timing shown in Figure 1-14. Ideally, all AGU adder results should be stable by the end of Q1 (for reads and stack writes) or Q3 (for writes or stack reads). Effective address selection should occur on rising Q2 or Q4. The W address register update (when required) should occur during Q2.

Alternatively, each AGU could be built as an asynchronous block allowing the address calculation and selection to ripple through. However, it is highly likely that this will result in many spurious address transitions which could effect power consumption if allowed to propagate too far.



Bit Reversed Address			
A0	A1	A2	A3
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	0
0	0	1	0
1	0	1	0
0	1	1	0
1	1	1	0
0	0	0	1
1	0	0	1
0	1	0	1
1	1	0	1
0	0	1	1
1	0	1	1
0	1	1	1
1	1	1	1

TABLE 4-7: BIT REVERSED ADDRESS SEQUENCE (16-ENTRY)

#### 4.5.1 Bit Reversed Addressing Implementation

Bit reversed addressing is only supported by the X AGU. The address adder carry reverse signal (see Figure 4-40) is asserted when:

1. XWB (W register selection) in the XMOD register is any value other than 15 (it is assumed that nobody will ever want to bit reverse address the stack) **and**
2. the BREN bit is set in the XBREV register **and**
3. the addressing mode is register in direct with post-increment

XB<14:0> is the bit reversed address modifier which is typically a constant, indirectly representing the size of the FFT data buffer. The XB values required to provide the correct bit reversal 'pivot' points for various size buffers are shown in Table 4-8.

**Note:** All bit reversed EA calculations assume word size data (LS-bit of every EA is always clear). The XB value is scaled accordingly to generate compatible (byte) addresses

Buffer Size (words)	12-bit Bit Reversed Address Modifier (XB)	XB Scaled for Word Sized Data
32768	0x4000	0x8000
16384	0x2000	0x4000
8192	0x1000	0x2000
4096	0x0800	0x1000
2048	0x0400	0x0800
1024	0x0200	0x0400
512	0x0100	0x0200
256	0x0080	0x0100
128	0x0040	0x0080
64	0x0020	0x0040
32	0x0010	0x0020
16	0x0008	0x0010
8	0x0004	0x0008
4	0x0002	0x0004
2	0x0001	0x0002

TABLE 4-8: ADDRESS MODIFIER VALUES

As can be seen from Figure 4-45, requiring that both the address modifier (constant) and the address in the W pointer are always in bit reversed format simplifies the hardware. Adding two bit reversed values though the adder with carry reversed enabled, will produce the correct bit reversed result.

When enabled, bit reversed addressing will only be executed with register indirect with post increment addressing and word sized data. It will not function for

all other addressing modes or byte sized data (normal addresses will be generated). When bit reversed addressing is active, the W address pointer will always be added to the address modifier (XB) and the offset associated with the register indirect addressing mode will be ignored. In addition, as word sized data is a requirement, the LS-bit of the EA is ignored (and always clear). An example word swap using bit reversed addressing is:



# **REGISTER 4-6: XBREV, X AGU BIT REVERSAL ADDRESSING CONTROL REGISTER (XXXXh)**

Upper Half:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
BREN	XB14	XB13	XB12	XB11	XB10	XB9	XB8
bit 15				bit 8			

Lower Half:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
XB7	XB6	XB5	XB4	XB3	XB2	XB1	XB0
bit 7				bit 0			

- bit 15 **BREN:** Bit Reversed Addressing (X AGU) Enable  
1 = Bit Reversed Addressing enabled  
0 = Bit Reversed Addressing disabled
- bit 14-0 **XB<14:0>:** X AGU bit reversed Modifier  
e.g. XB<14:0> = 0x0080; modifier for a 128 point radix-2 FFT

Legend			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	1 = bit is set	0 = bit is cleared	x = bit is unknown



## 4.6 Data Addressing in Program Space

Many applications require significant amounts of fixed data (e.g. MELP) which can only be held in non-volatile memory. This data can also exceed the 32K word limit of data space memory. Consequently, this data will have to reside in on-chip program FLASH, ROM or in external program space. In order to accommodate this requirement, two addressing options are provided.

1. The table instructions allows direct movement of word and byte data respectively between program and data space without passing through an intermediate register.
2. The upper part of data space may be configured to map into a 16K word segment of program space.

The operation of these addressing options is discussed in Section 1.2 and Section 1.3. The following sections revisit the table instructions, in particular the addressing modes supported.

#### 4.6.1 Table Instruction Operation

There are 4 'table' instructions as shown in Table 4-9 which operate with MODE2 addressing modes for both operand source and destination. They operate in a manner similar to that for data space access except that the EA for program space (source or destination) is concatenated with a 8-bit page register, TABPAG<7:0> to create a 24-bit address as shown in Figure 1-8. All table instructions treat the program memory as 16-bit wide, byte addressable (i.e. same as data space). Program space EA[24:1] forms the 24-bit program memory address and the EA[0] becomes a byte select bit. The TBLRDL and TBLWTL instructions are dedicated to accessing the LS program word.

The program word is viewed as a 32-bit entity which consists of a 24-bit program word plus an 8-bit 'phantom' byte (MS-byte). This allows TBLRDH and TBLWTH instructions (which are dedicated to accessing the MS program word) to maintain orthogonality with TBLRDL and TBLWTL. For TBLRDH and TBLWTH instructions, EA[0] remains a byte select bit but physical memory is only present in the LS-byte (EA[0]=0). A byte read of the MS-byte (EA[0]=1) will return 0x00.

#### 4.6.1.1 Table Read Operation

The program memory is always read as 24-bit long words. The LS-bit of the EA is used by the TBLRDL and TBLWTL (if required) to select required byte of the LS program word. Table 4-9 indicates which instruction and data width will access the various parts of the program word.

TBLRDH.w reads a data word from  $[EA_{src}]_{<31:16>}$ , though  $[EA_{src}]_{<31:24>}$  will equal 0x00. TBLRDH.b reads a data byte from  $[EA_{src}]_{<31:24>}$  (always equal

to 0x00) or  $[EA_{src}]_{<16:23>}$  based on the state of  $EA[0]$ . The data byte is transferred into destination  $EA[7:0]$ .

TBLRDL.w reads a data word from [EA<sub>src</sub>]<sub><15:0></sub>.  
TBLRDL.b reads a data byte from [EA<sub>src</sub>]<sub><15:0></sub> or [EA<sub>src</sub>]<sub><7:0></sub> based on the state of EA[0]. The data byte is transferred into destination EA[7:0].

For most applications, it is assumed that only the LS word of the program word will be used for data storage. The MS byte of the program word would then typically contain an illegal instruction trap to prevent the machine from ever inadvertently attempting to execute data. However, TBLRDH is provided to allow the use of all program memory for data storage if desired.

#### 4.6.1.2 Table Writes

Refer to the Program Memory DOS-00204 for details about table write operation.

Instruction	EA[0]	Program Space Data Move Operation	
		Source	Destination
TBLRDH.w <sup>†</sup>	x	[EA <sub>src</sub> ]<31:16>	[EA <sub>dst</sub> ]<15:0>
TBLRDH.b	0	[EA <sub>src</sub> ]<16:23>	[EA <sub>dst</sub> ]<7:0>
TBLRDH.b <sup>†</sup>	1	[EA <sub>src</sub> ]<31:24>	[EA <sub>dst</sub> ]<7:0>
TBLRDL.w	x	[EA <sub>src</sub> ]<15:0>	[EA <sub>dst</sub> ]<15:0>
TBLRDL.b	0	[EA <sub>src</sub> ]<7:0>	[EA <sub>dst</sub> ]<7:0>
TBLRDL.b	1	[EA <sub>src</sub> ]<15:8>	[EA <sub>dst</sub> ]<7:0>

**Note 1:** MS-byte read will return 0x00

**TABLE 4-9: TABLE INSTRUCTION SUMMARY**

#### 4.6.2 MODE 2 Addressing for Program Space

MODE2 determines the addressing mode for the operand and source/destination in program space or the operand and source/destination from data space, depending upon instruction requirements. It follows the same definition for each encoding as MODE1 except that it applies to only one operand. The MODE1 signed 5-bit constant value mode makes little sense where MODE2 is used, and is therefore not supported.

In summary, MODE2 for program space data accesses supports the addressing mode shown in Table 4-10.

MODE2 submode 0 is meaningless for TBLRD source and TBLWT destination operands as the program memory must be addressed with a pointer.

The following addressing mode descriptions are for table read operations.

MODE2 Bit Encoding	Function (Source)	Function (Destination)	Description
000	EA = Wsrc <sup>1</sup>	EA = Wdst <sup>2</sup>	Register direct
001	EA = [Wsrc]	EA = [Wdst]	Register indirect
010	EA = [Wsrc]-= 1	EA = [Wdst]-= 1	Register indirect post-decremented
011	EA = [Wsrc]+= 1	EA = [Wdst]+= 1	Register indirect post-incremented
100	EA = [Wsrc-=1]	EA = [Wdst-=1]	Register indirect pre-decremented
101	EA = [Wsrc+=1]	EA = [Wdst+=1]	Register indirect pre-incremented
110	Unused	Unused	
111	Unused	Unused	

Note 1: Not meaningful for TBLRD instructions

Note 2: Not meaningful for TBLWT instructions

TABLE 4-10:MODE 2 ADDRESSING MODE DEFINITION (PROGRAM SPACE)

#### 4.6.2.1 Mode2, Register Direct

Addressing MODE2, submode 0 is register direct. The implied effective address is the memory mapped address of register Wdst.

The table read result is written to Wdst as shown in Figure 4-48. Wdst is accessed through addressing its memory mapped image.

Note that register direct for the operand source of a table read, and the operand destination for a table write has no meaning. The X AGU would generate an

EA which would address the memory mapped version of Wsrc or Wdst. When concatenated with the TABPAG register, this address will be the same but within a program space page.

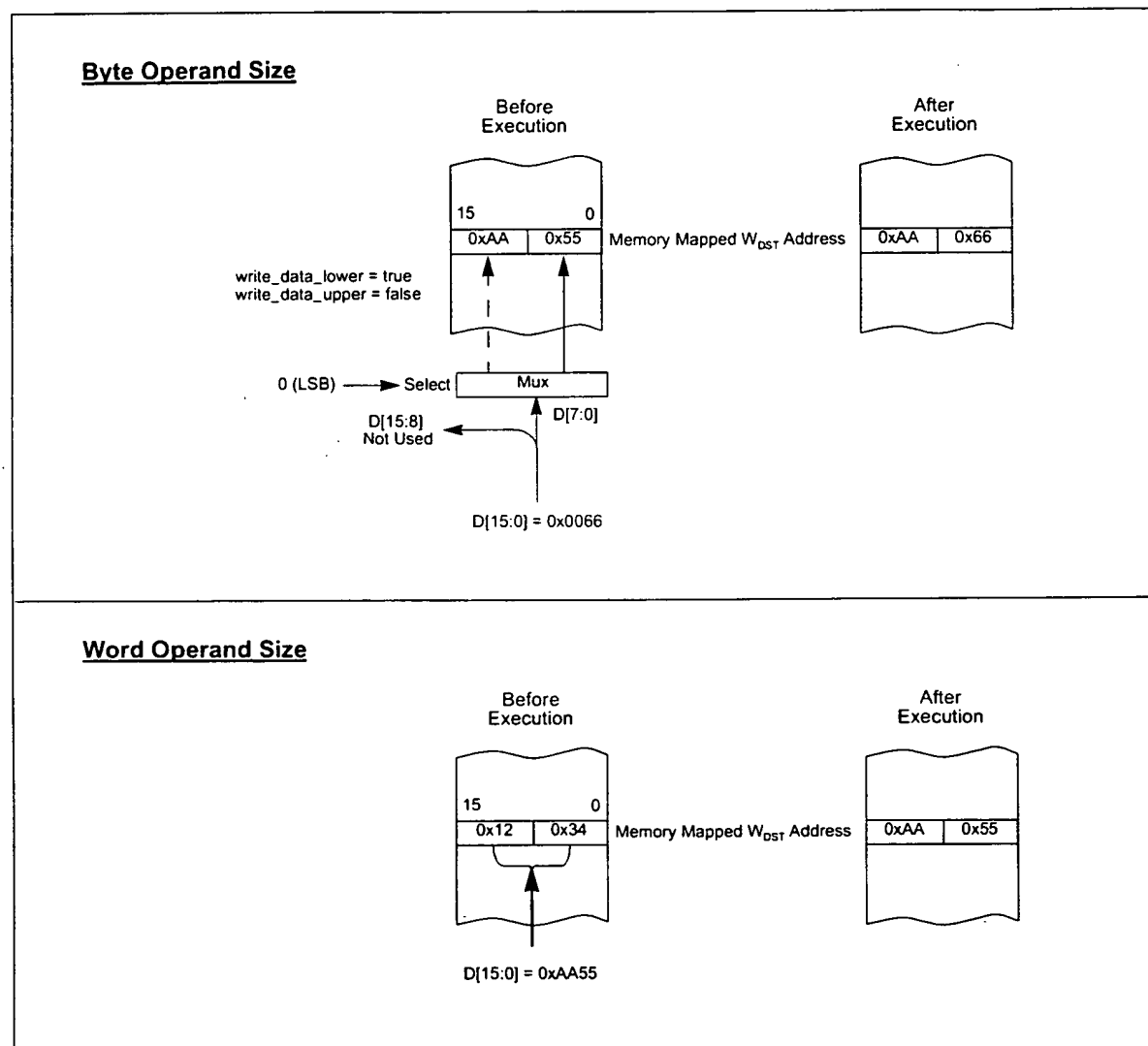


FIGURE 4-48: REGISTER DIRECT, TABLE READ OPERAND DESTINATION (MODE2, SUBMODE 0)

#### 4.6.2.2 Mode2, Register Indirect

Addressing MODE2, submode 1 is register indirect. The effective address contained in register Wsrc points to the operand as shown in Figure 4-49, or Wdst points to the result destination as shown in

Figure 4-50. For table read instructions, TABPAG<7:0> is concatenated onto the source EA to form the 24-bit program space EA.

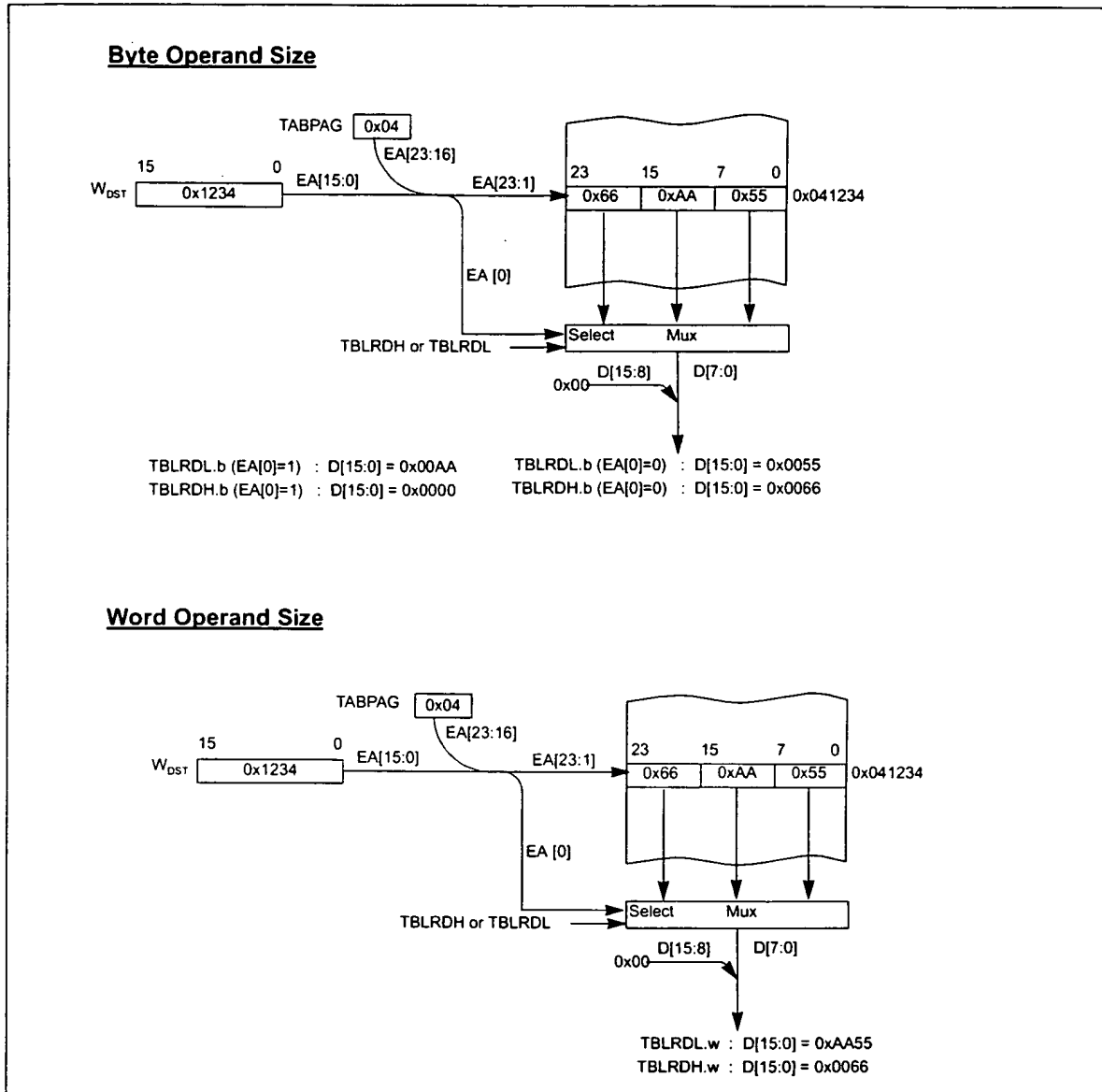


FIGURE 4-49: REGISTER INDIRECT, TABLE READ OPERAND SOURCE (MODE2, SUBMODE 1)

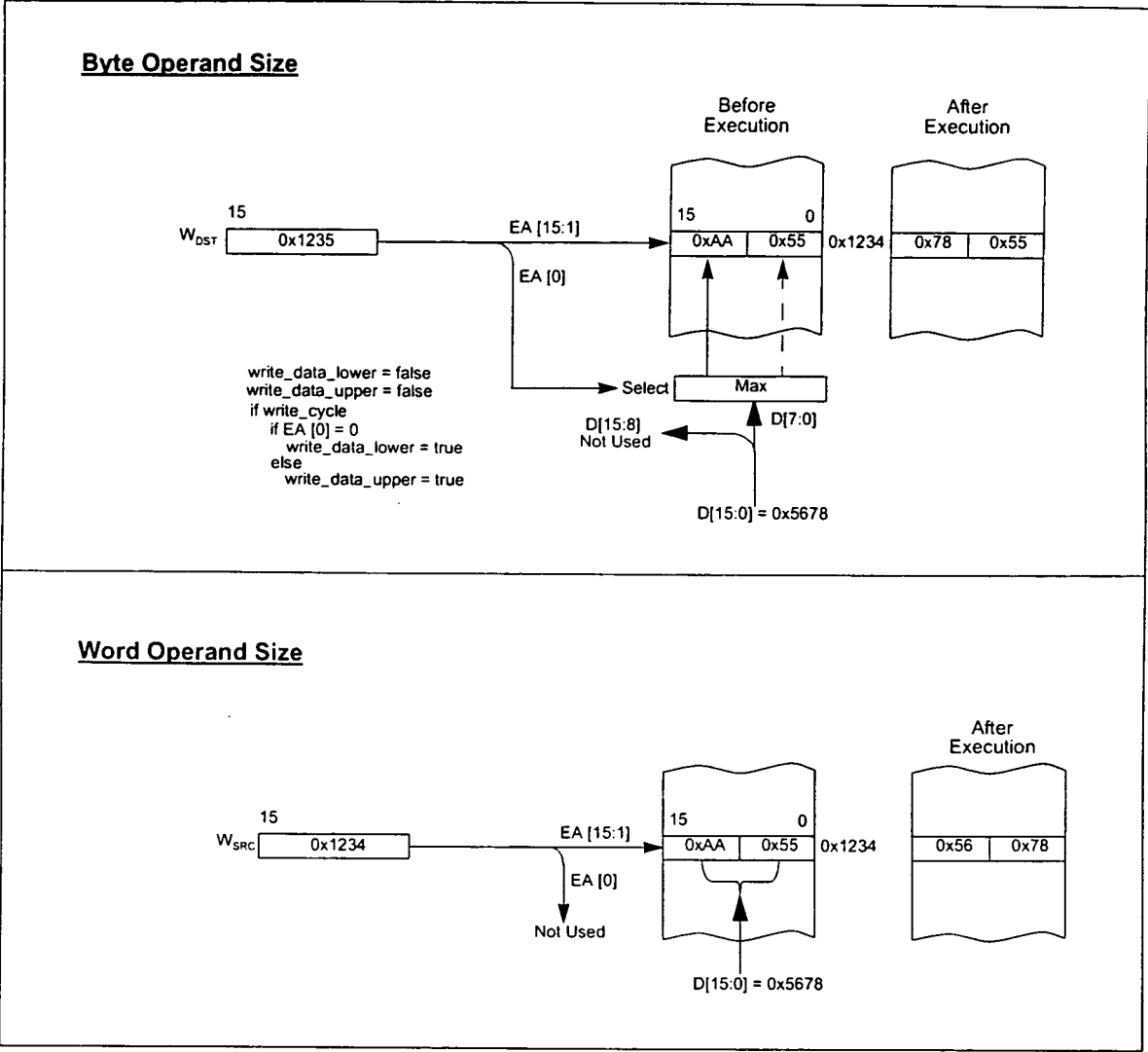


FIGURE 4-50: REGISTER INDIRECT, TABLE READ RESULT DESTINATION (MODE2, SUBMODE 1)

#### 4.6.2.3 Mode2, Register Indirect with Post Decrement

Addressing MODE2, submode 2 is register indirect with post decrement. The effective address contained in register Wsrc points to the operand, or the effective address contained in register Wdst points to the result destination.

Wsrc or Wdst is then post decremented as shown in Figure 4-51 and Figure 4-52.

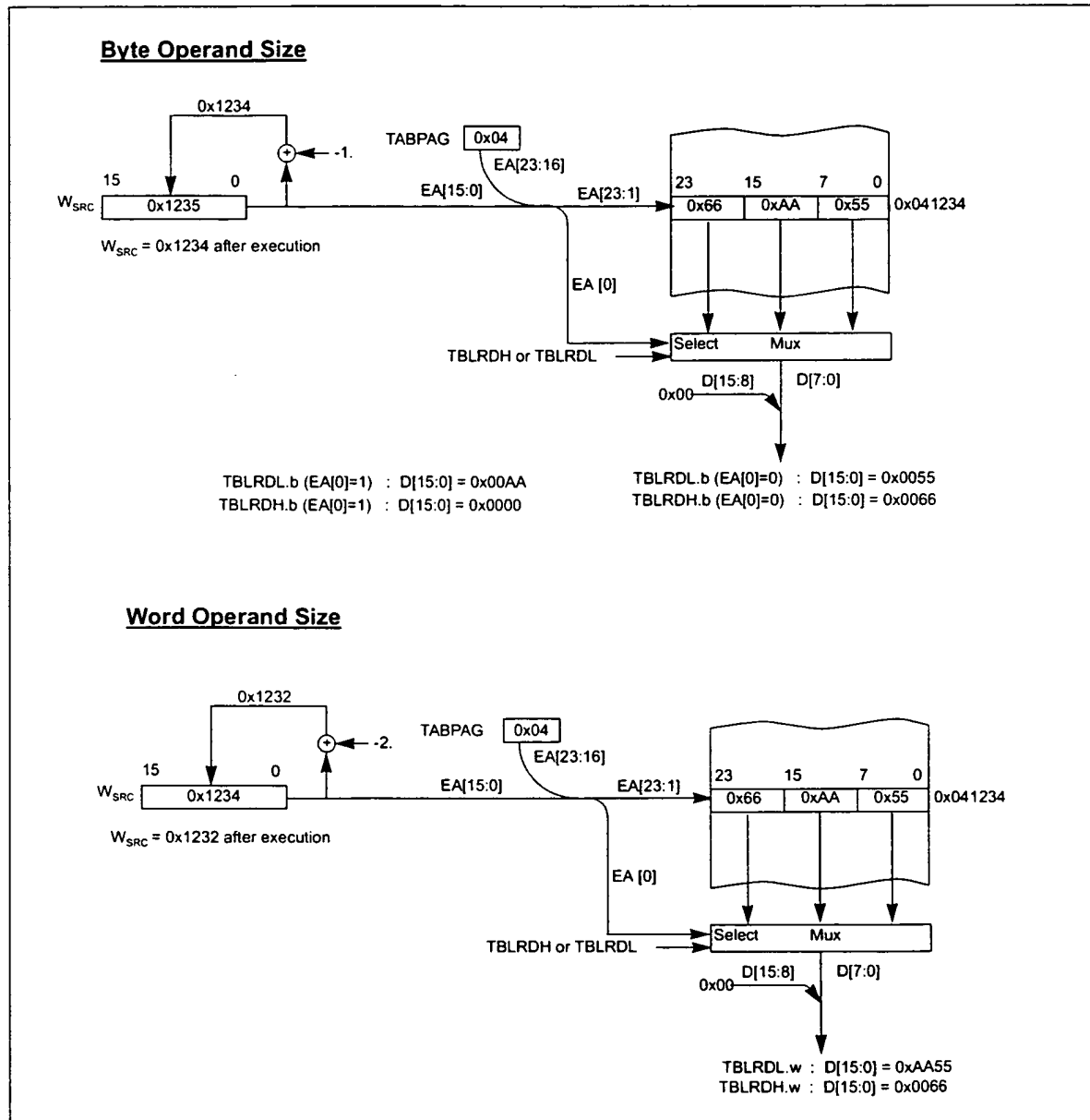
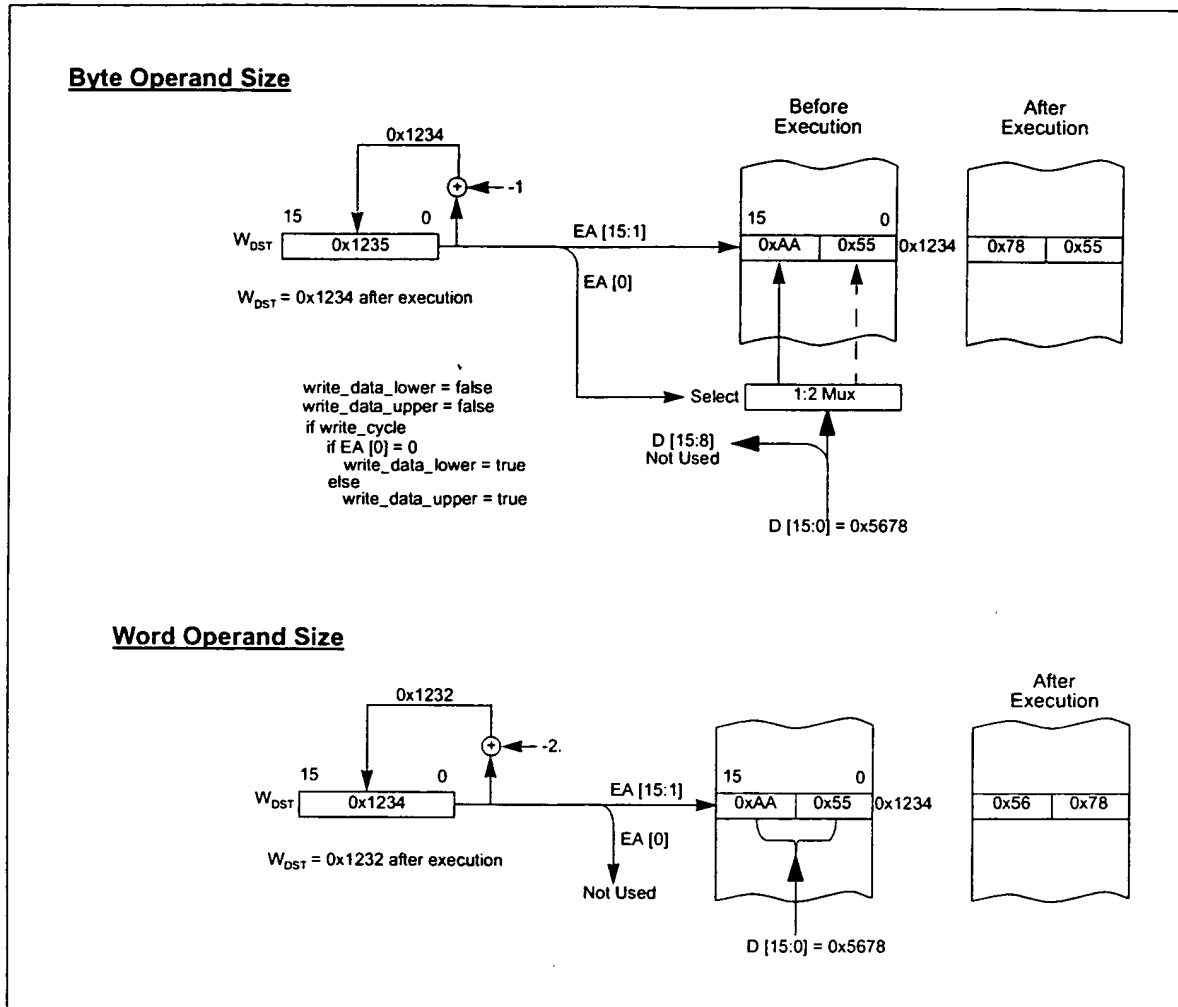


FIGURE 4-51: REGISTER INDIRECT WITH POST DECREMENT, TABLE READ SOURCE OPERAND (MODE2, SUBMODE 2)



**FIGURE 4-52: REGISTER INDIRECT WITH POST DECREMENT, TABLE READ RESULT DESTINATION (MODE2, SUBMODE 2)**

#### 4.6.2.4 Mode2, Register Indirect with Post Increment

Wsrc or Wdst are then incremented as shown in Figure 4-53 and Figure 4-54.

Addressing MODE2, submode 3 is register indirect with post increment. The effective address contained in register Wsrc points to the source operand, or the effective address contained in register Wdst points to the result destination.

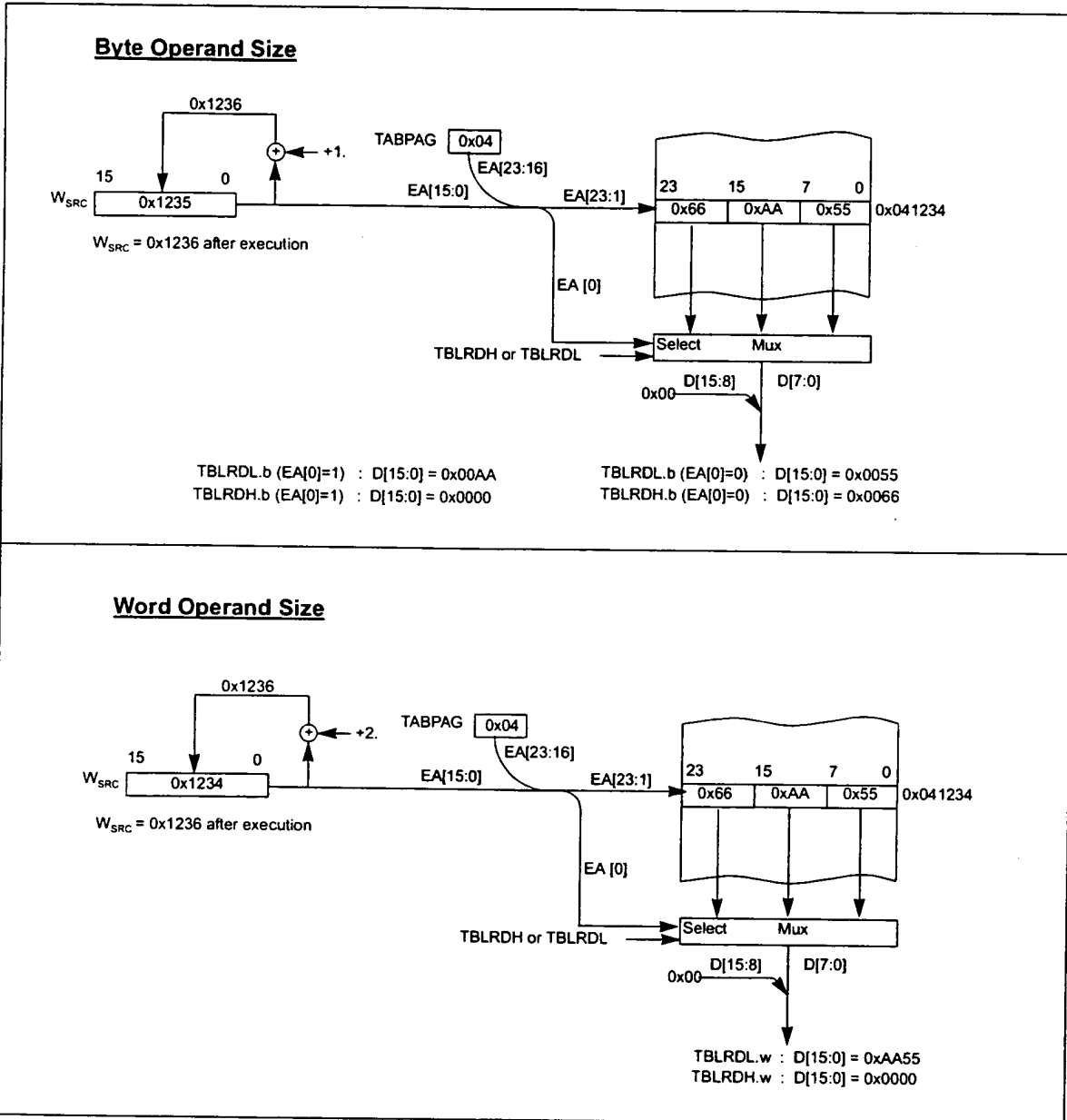


FIGURE 4-53: REGISTER INDIRECT WITH POST INCREMENT, TABLE READ OPERAND SOURCE (MODE2, SUBMODE 3)



09870457-060404

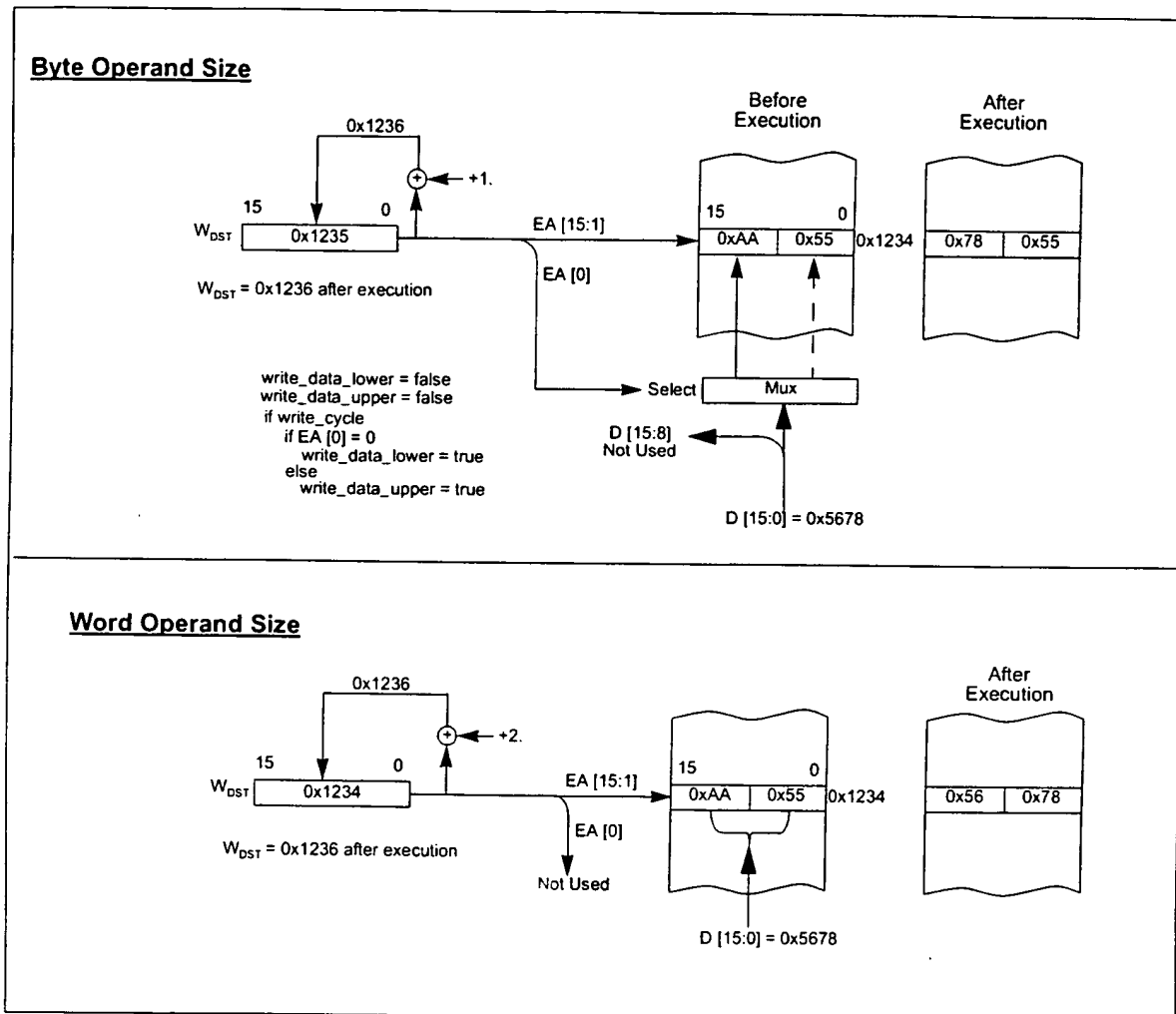


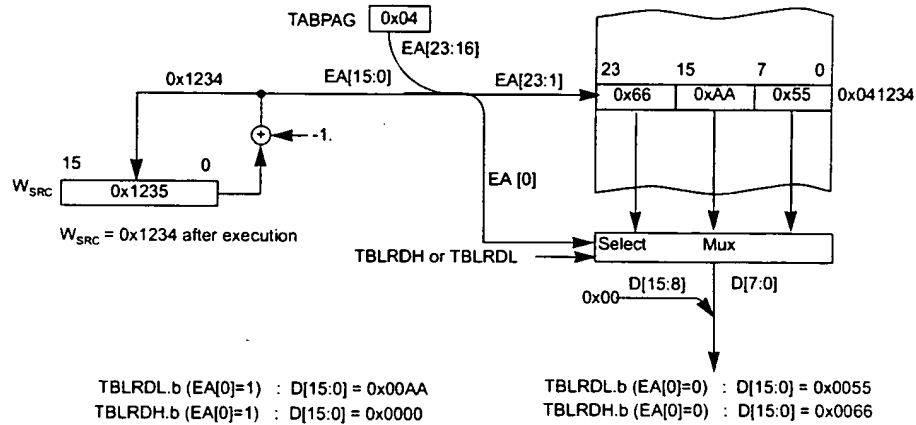
FIGURE 4-54: REGISTER INDIRECT WITH POST INCREMENT, TABLE READ RESULT DESTINATION (MODE2, SUBMODE 3)

#### 4.6.2.5 Mode2, Register Indirect with Pre Decrement

Addressing MODE2, submode 4 is register indirect with pre decrement.

Register Wsrc or Wdst is decremented to form the effective address which points to the operand as shown in Figure 4-57 and Figure 4-58.

##### Byte Operand Size



##### Word Operand Size

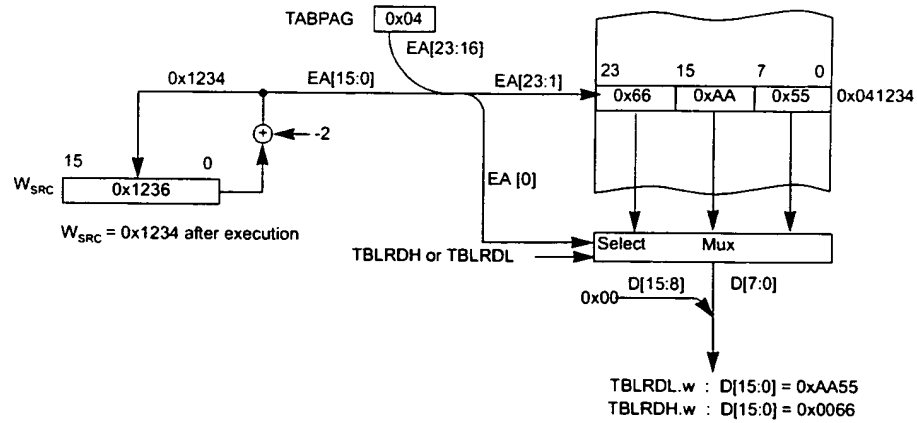
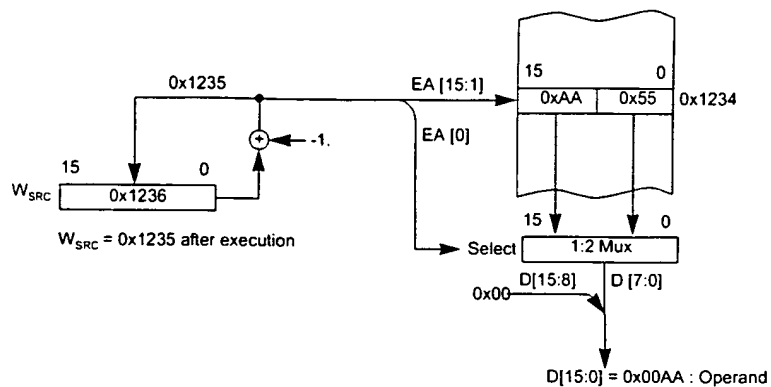


FIGURE 4-55: REGISTER INDIRECT WITH PRE DECREMENT, TABLE READ SOURCE OPERAND (MODE2, SUBMODE 4)

### Byte Operand Size



### Word Operand Size

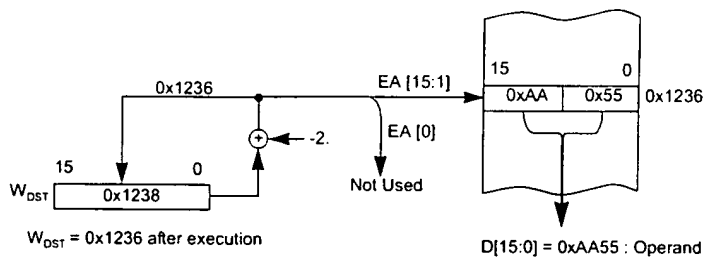


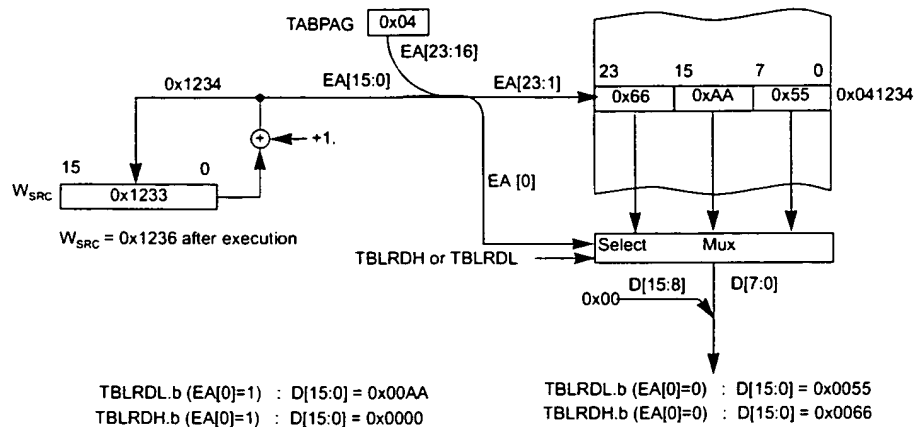
FIGURE 4-56: REGISTER INDIRECT WITH PRE DECREMENT, TABLE READ RESULT DESTINATION (MODE2, SUBMODE 4)

#### 4.6.2.6 Mode2, Register Indirect with Pre Increment

Addressing MODE2, submode 5 is register indirect with pre increment.

Register Wsrc or Wdst is incremented to form the effective address which points to the operand as shown in Figure 4-57 and Figure 4-58.

##### Byte Operand Size



##### Word Operand Size

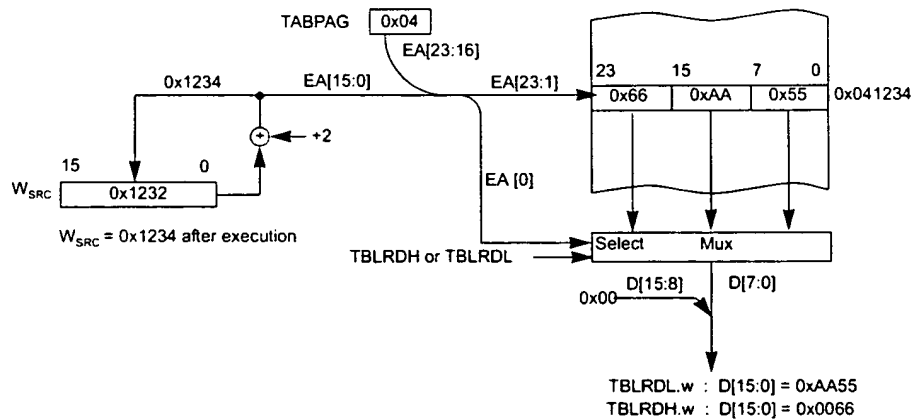
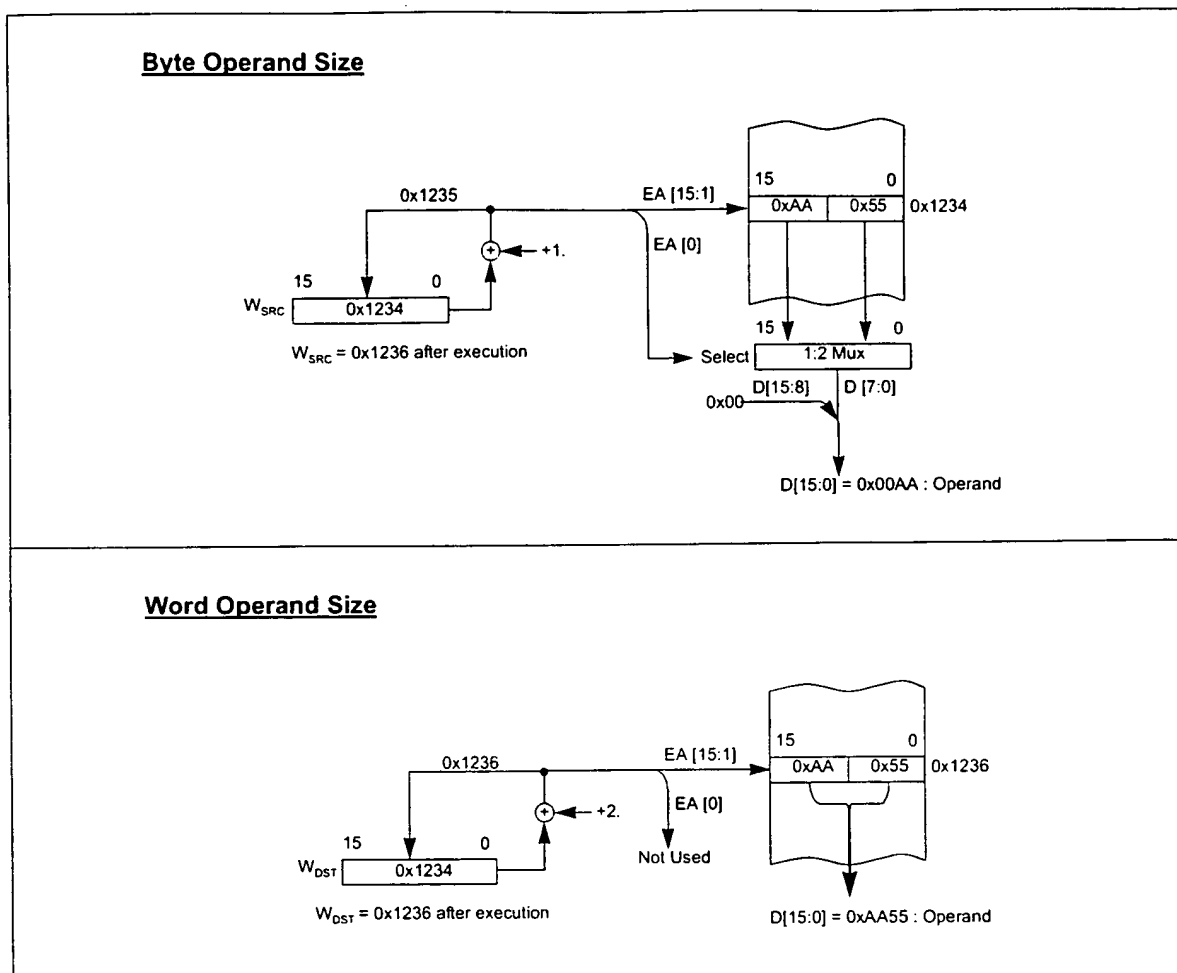


FIGURE 4-57: REGISTER INDIRECT WITH PRE INCREMENT, TABLE READ SOURCE OPERAND (MODE2, SUBMODE 5)



**FIGURE 4-58: REGISTER INDIRECT WITH PRE INCREMENT, TABLE READ RESULT DESTINATION (MODE2, SUBMODE 5)**

APPENDIX C

09870457 . 060101

FIGURE 0-1: FLOW DIAGRAM XOR, SUBR, SUBBR, SUBB, SUB, MOV, IOR, AND, ADDC, ADD

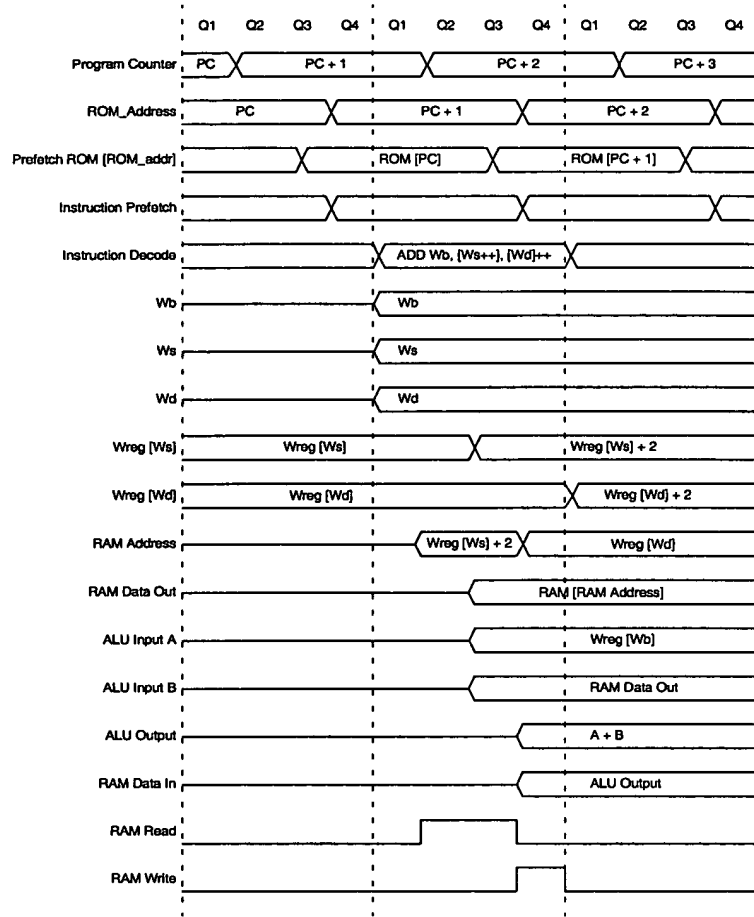


FIGURE 0-2: FLOW DIAGRAM XORLS, SUBRLS, SUBLS, SUBBRLS, SUBBLS, IORLS, ANDLS, ADDLS, ADDCLS

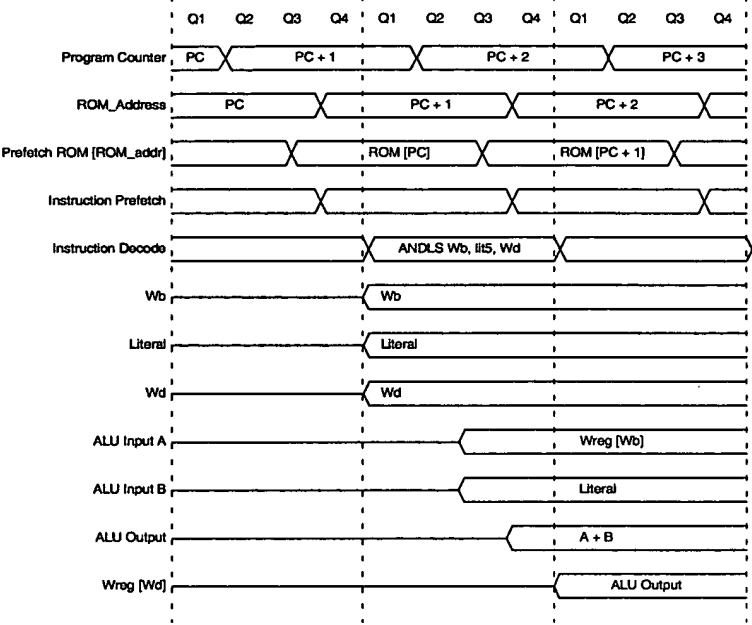






FIGURE 0-4: FLOW DIAGRAM ASR, LSR, ZE, SE, SL, RLC, RLNC, RRC, RRNC

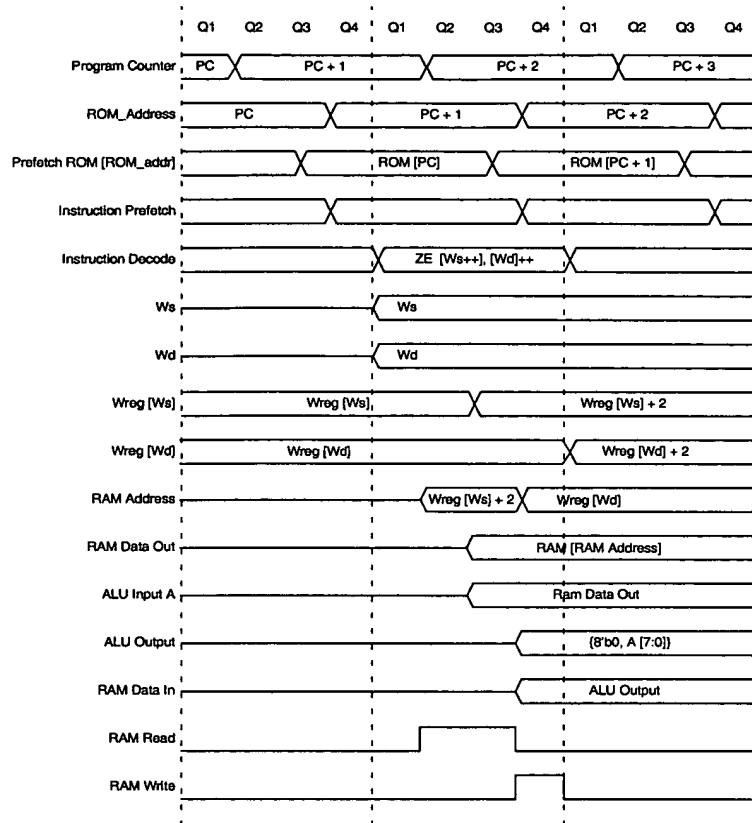


FIGURE 0-5: FLOW DIAGRAM CPB, CP

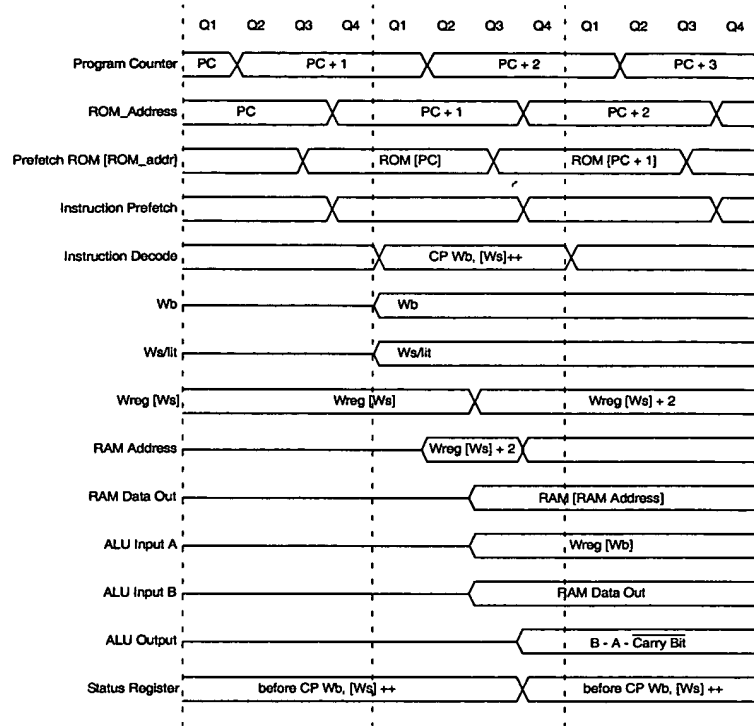
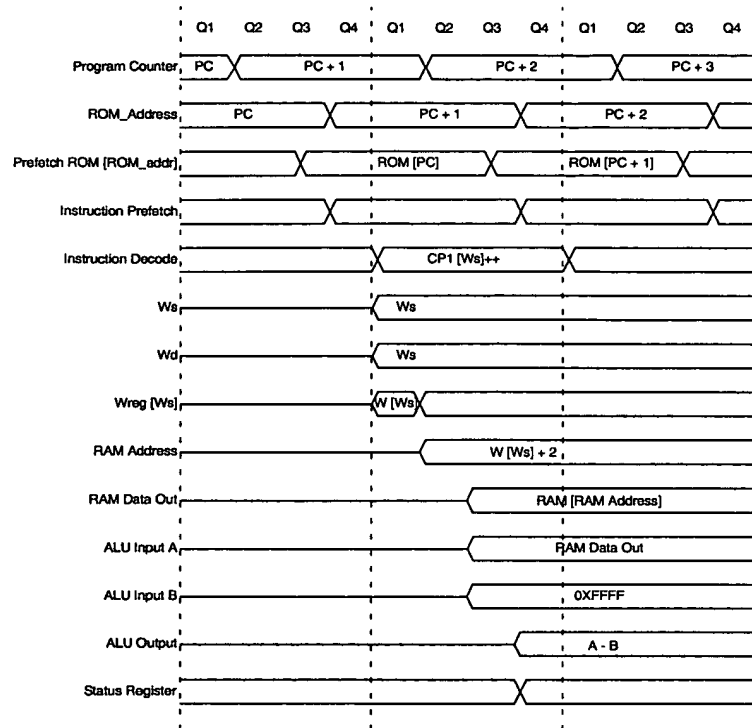


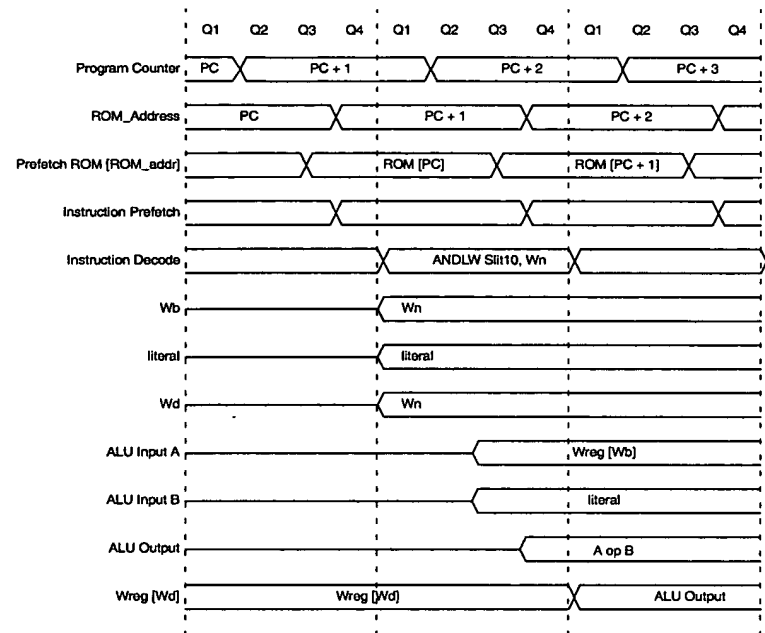
FIGURE 0-6: FLOW DIAGRAM CP1, CP0



The diagram illustrates the timing of the CP instruction across four clock cycles (Q1, Q2, Q3, Q4). The events are as follows:

- Program Counter:** Updates from PC to PC+1 at the start of Q2, from PC+1 to PC+2 at the start of Q3, and from PC+2 to PC+3 at the start of Q4.
- ROM\_Address:** Updates from PC to PC+1 at the start of Q2, and from PC+1 to PC+2 at the start of Q3.
- Prefetch ROM [ROM\_addr]:** ROM[PC] is fetched in Q2, and ROM[PC+1] is fetched in Q3.
- Instruction Prefetch:** The instruction is prefetched in Q2.
- Instruction Decode:** The instruction is decoded in Q2, identifying CP Wb, lit5.
- Wb:** The write-back address Wb is determined in Q2.
- lit5:** The literal value lit5 is determined in Q2.
- Wreg [Wb]:** The register value Wreg[Wb] is determined in Q2.
- ALU Input A:** The ALU input A is Wreg[Wb], determined in Q2.
- ALU Input B:** The ALU input B is lit5, determined in Q2.
- ALU Output:** The ALU output is A - B, determined in Q2.
- Status Register:** The status register is updated from "before CP Wb, lit5" to "after CP Wb, lit5" in Q2.

FIGURE 0-8: FLOW DIAGRAM XORLW, SUBLW, SUBBLW, MOVLW, MOVL, IORLW, ANDLW, ADDLW, ADDCLW



09370457-060101



FIGURE 0-10: FLOW DIAGRAM CPFB, CPF1, CPF0, CPF

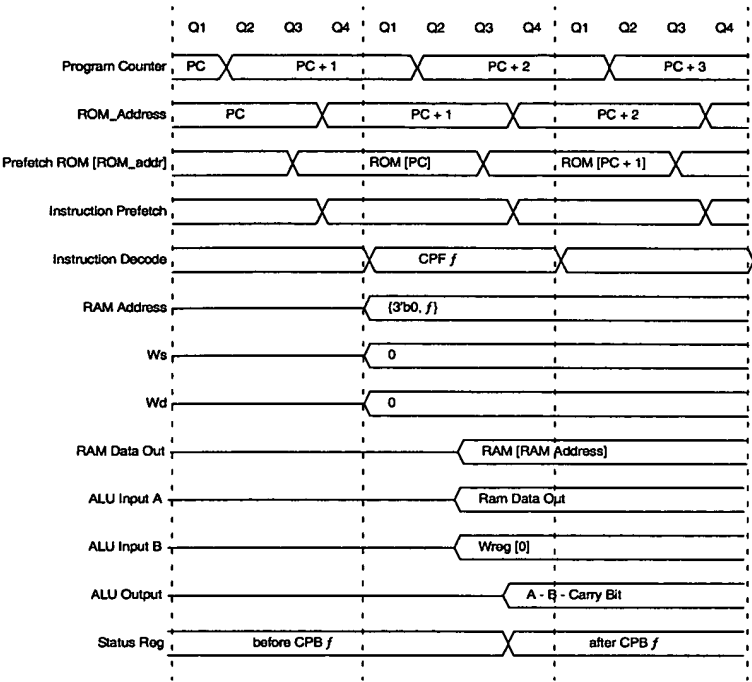
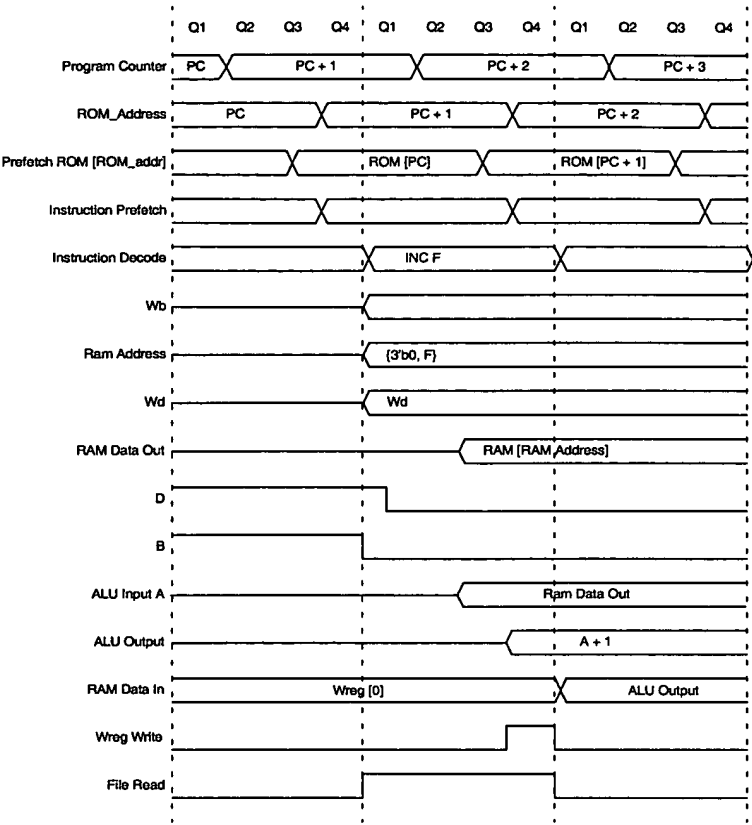


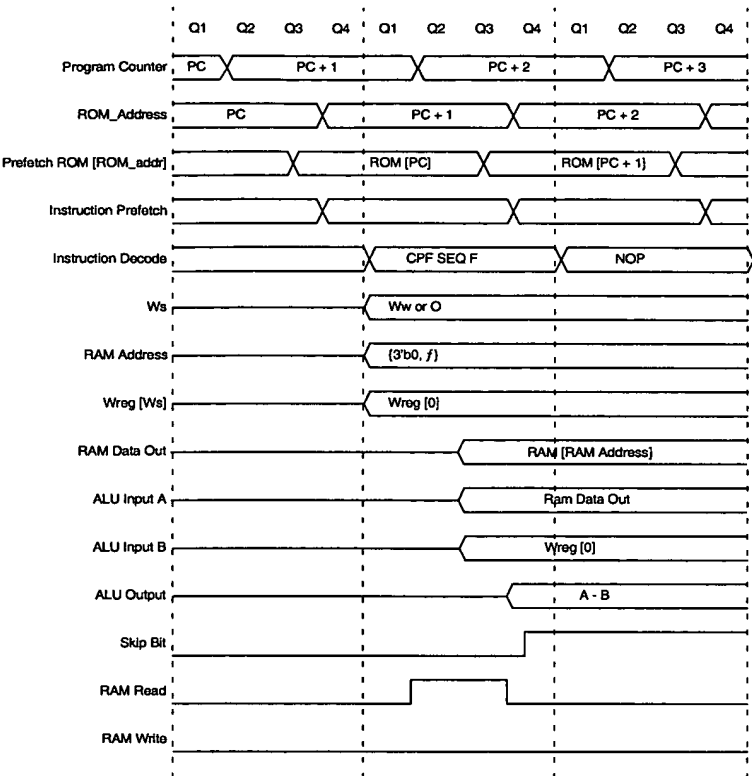


FIGURE 0-11: FLOW DIAGRAM INCF, DECF, NEGF, SETF, COMF, CLRF



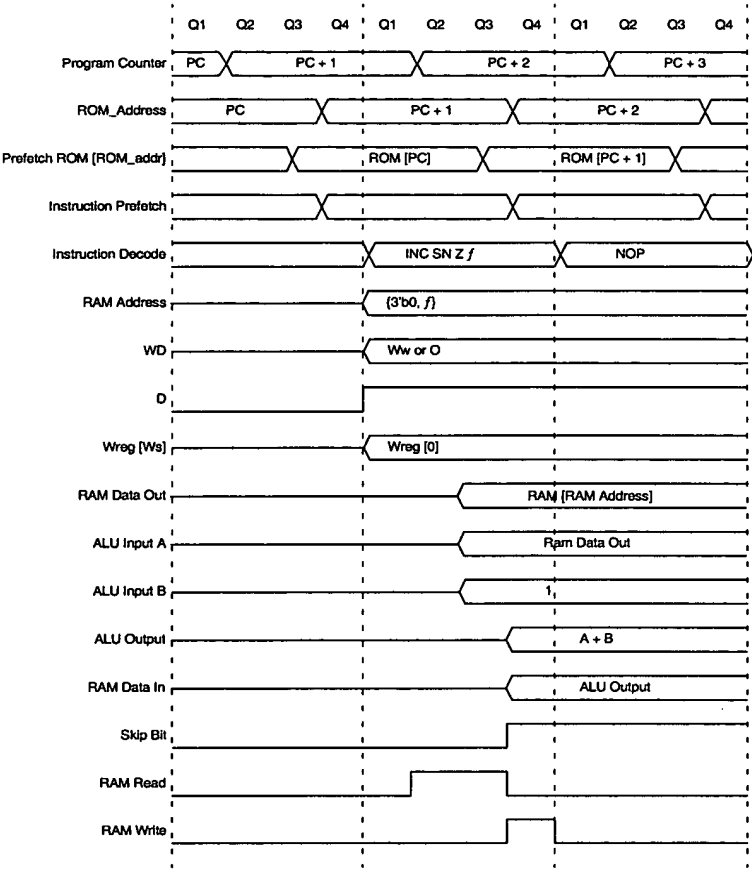
09870457-060101

FIGURE 0-12: FLOW DIAGRAM CPFSEQ, FPFSGT, CPFSLT, CPFSNE



09870457-060101

FIGURE 0-13: FLOW DIAGRAM INCFSNZ, INCFSZ, DECFSNZ, DECFSZ



The diagram illustrates the timing of the SWAP instruction across four clock cycles (Q1, Q2, Q3, Q4). The events are as follows:

- Program Counter:** Updates from PC to PC+1 at the start of Q1, PC+1 to PC+2 at the start of Q2, and PC+2 to PC+3 at the start of Q3.
- ROM\_Address:** Updates from PC to PC+1 at the start of Q1, and PC+1 to PC+2 at the start of Q2.
- Prefetch ROM [ROM\_addr]:** ROM[PC] is fetched during Q1, and ROM[PC+1] is fetched during Q2.
- Instruction Prefetch:** The instruction is prefetched during Q1.
- Instruction Decode:** The instruction is decoded during Q2, identifying the SWAP Wn operation.
- Wb:** The register index Wn is decoded during Q2.
- Wd:** The register index Wn is decoded during Q2.
- B:** The branch condition is decoded during Q2.
- ALU Input A:** The register file outputs Wreg[Wn] to the ALU input A during Q3.
- ALU Output:** The ALU performs the swap operation on the input A, resulting in the output {A[7:0], A[15:8]} during Q3.
- Wreg [Wn]:** The register file outputs the ALU output back to the register file during Q4.

FIGURE 0-15: FLOW DIAGRAM STW

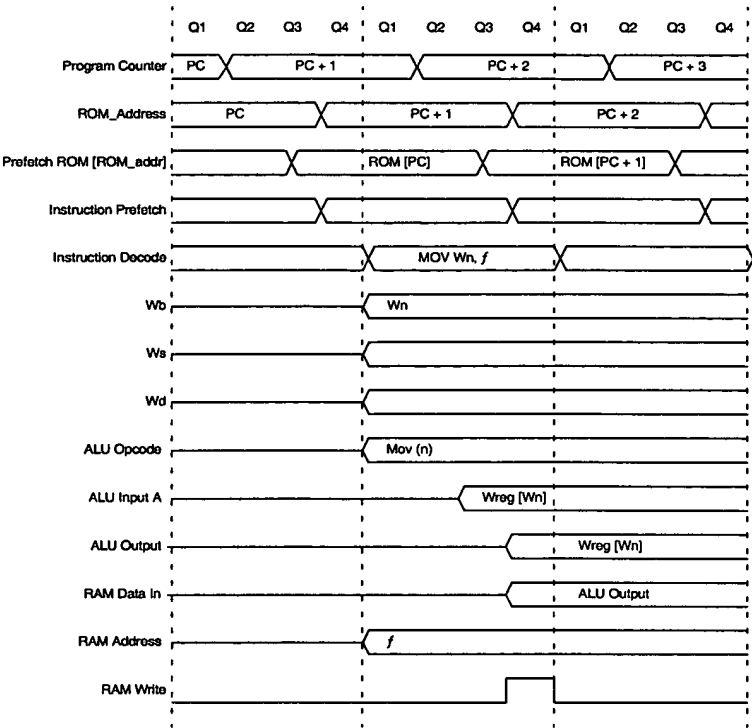
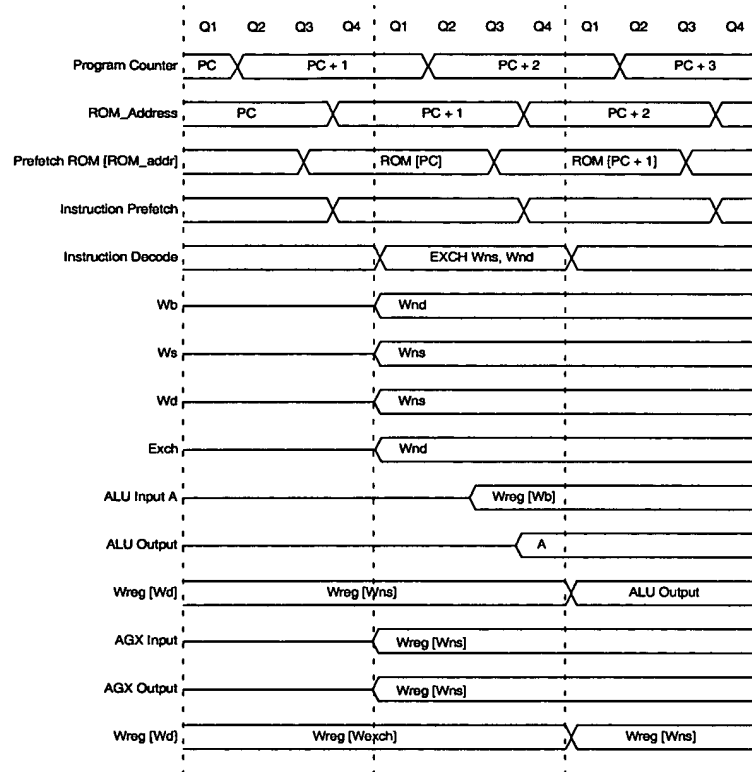


FIGURE 0-16: FLOW DIAGRAM EXCH



09870457 060401

The timing diagram illustrates the first instruction cycle of the TMS320C49 DSP. The horizontal axis is divided into quarters (Q1, Q2, Q3, Q4) for each of the next four quarters. The vertical axis lists the following components and their state during the cycle:

- Program Counter:** PC (Q1), PC + 1 (Q2), PC + 2 (Q3), PC + 3 (Q4).
- ROM\_Address:** PC (Q1), PC + 1 (Q2), PC + 2 (Q3), PC + 3 (Q4).
- Prefetch ROM [ROM\_addr]:** ROM [PC] (Q2), ROM [PC + 1] (Q3).
- Instruction Prefetch:** BSW.C [Ws], Wb (Q2).
- Instruction Decode:** BSW.C [Ws], Wb (Q2).
- Wb:** Wb (Q2).
- Ws:** Ws (Q2).
- Wd:** Ws (Q2).
- Wreg [Ws]:** Wreg [Ws] (Q2).
- RAM Address:** Wreg [Ws] (Q2).
- RAM Data Out:** RAM [RAM\_Address] (Q2).
- ALU Input A:** RAM Data Out (Q2).
- ALU Output:** A [Wb] ← C (Q2).
- RAM Data In:** ALU Output (Q2).
- RAM Write:** Active (Q2).
- RAM Read:** Active (Q2).

FIGURE 0-18: FLOW DIAGRAM BTSTW

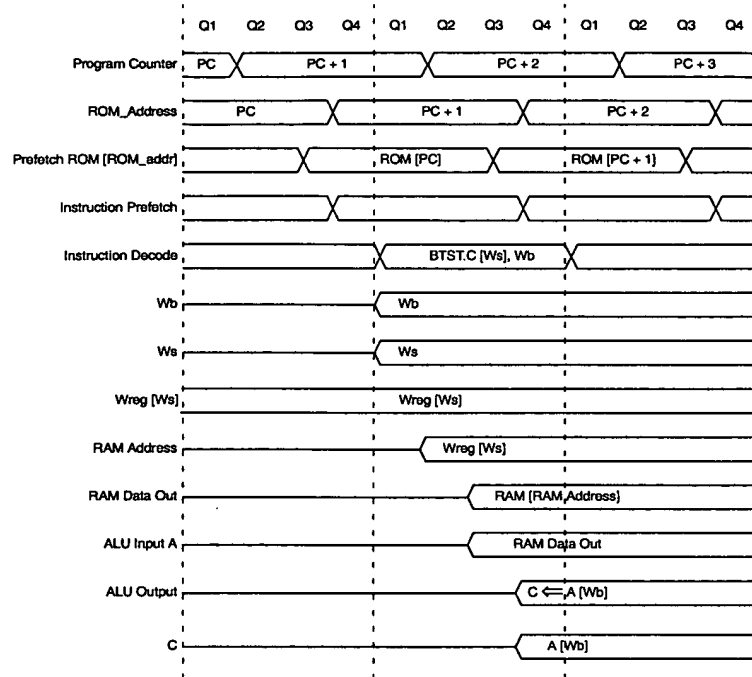




FIGURE 0-19: FLOW DIAGRAM BCLRF, BTSTSF, BTSTF, BTGF, BSETF

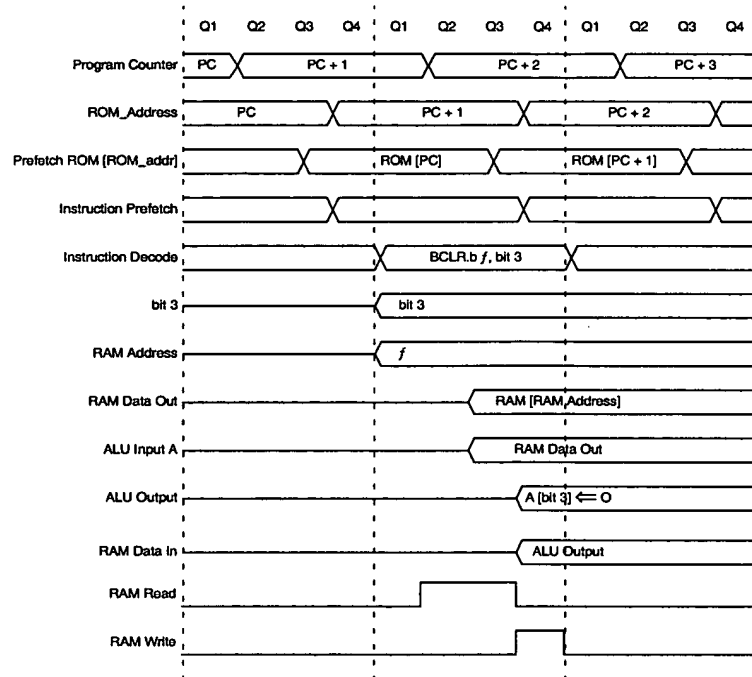


FIGURE 0-20: FLOW DIAGRAM BSET, BTG, BTST, BTSTS, BCLR

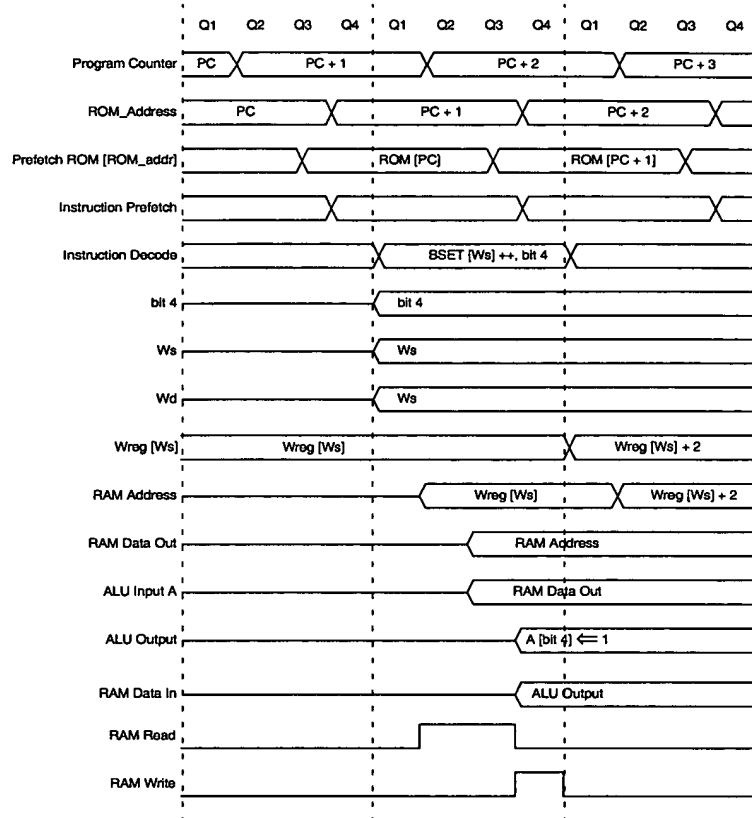


FIGURE 0-21: FLOW DIAGRAM BTSS, BTSC, BTFSC, BTFSS

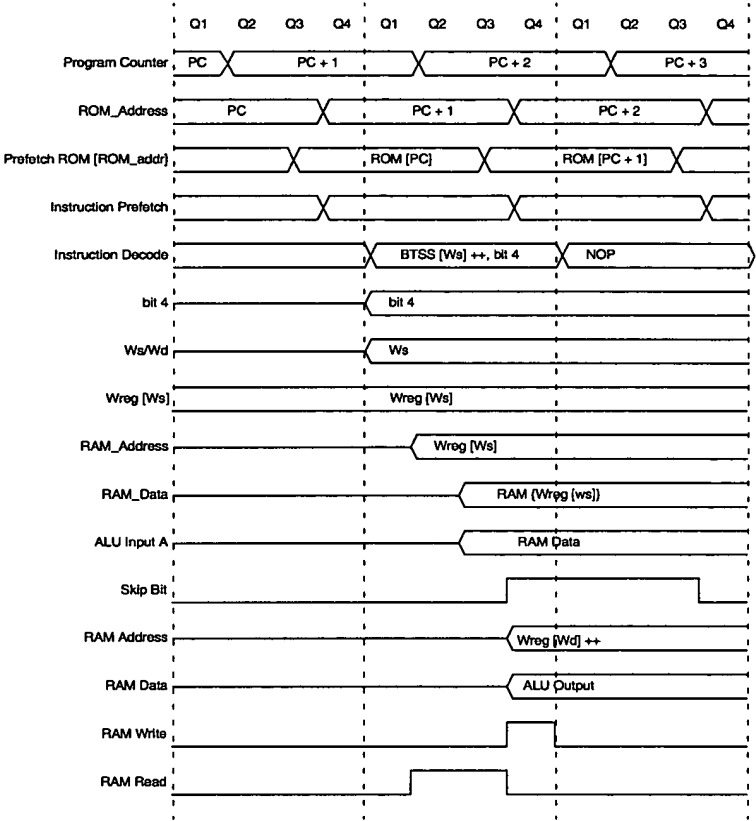
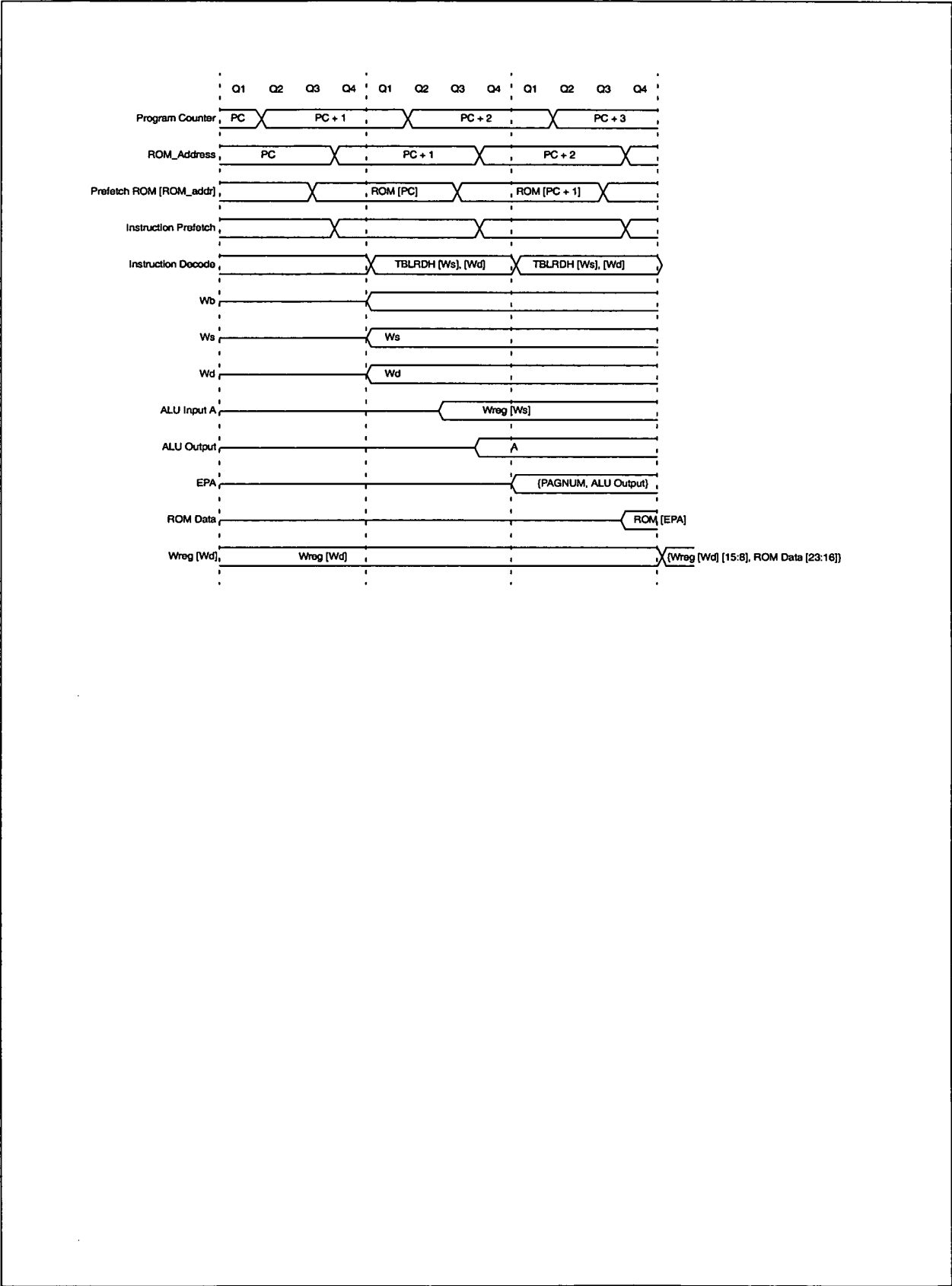


FIGURE 0-22: FLOW DIAGRAM TBLRDH, TBLRDL



**060907**

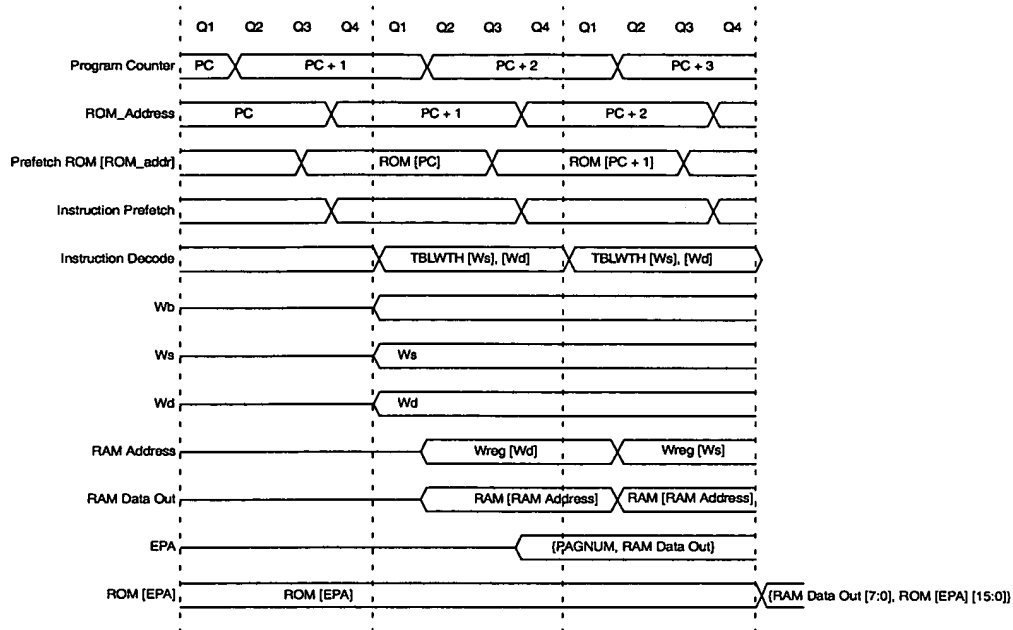
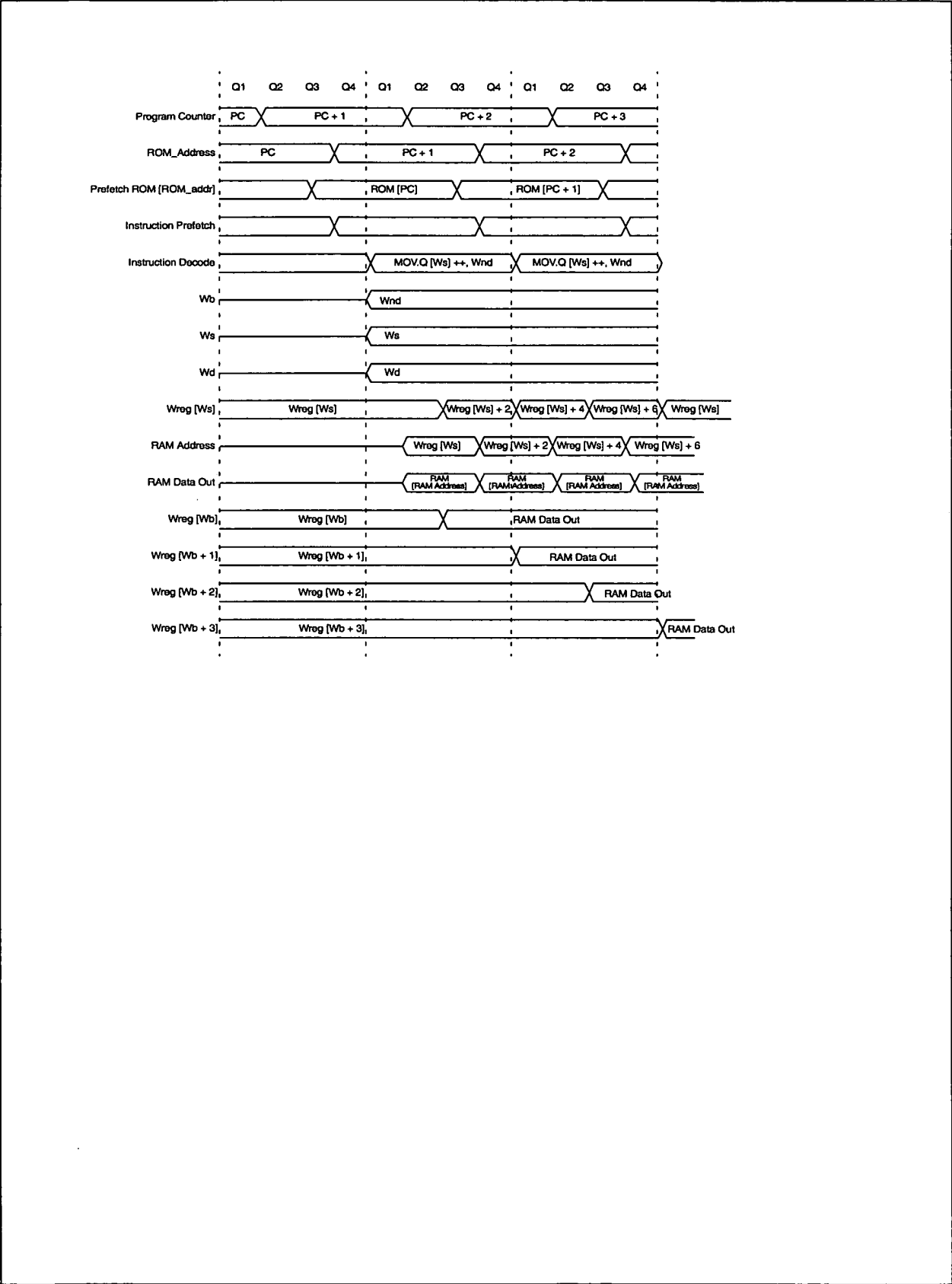


FIGURE 0-24: FLOW DIAGRAM LDQW



00070457 " 000404

**SECRET**

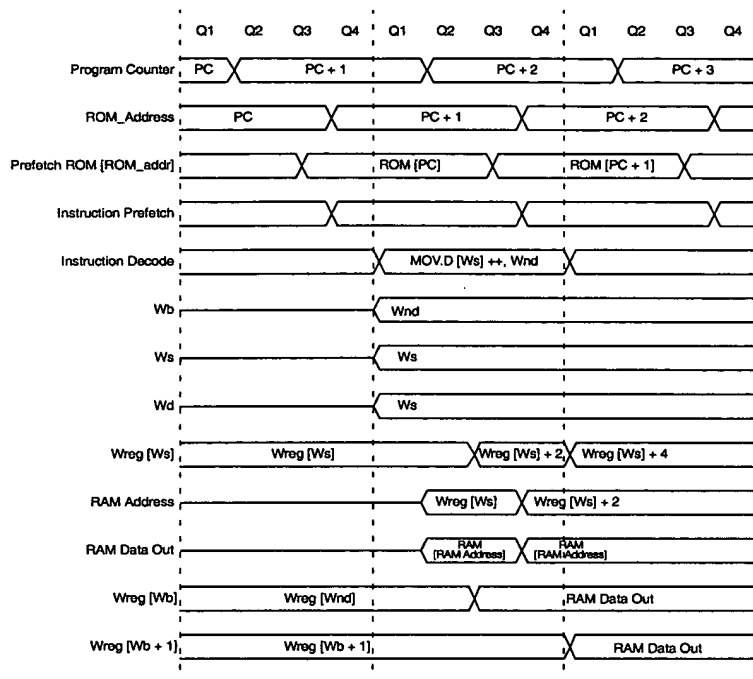






FIGURE 0-27: FLOW DIAGRAM STDW

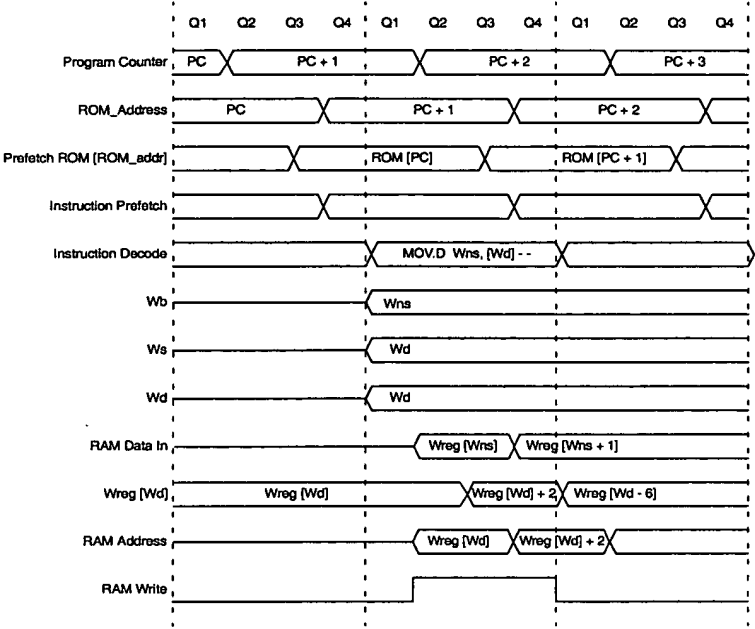


FIGURE 0-27: FLOW DIAGRAM STDW

FIGURE 0-28: FLOW DIAGRAM MULS, MULSU, MULSULS, MULU, MULULS, MULUS

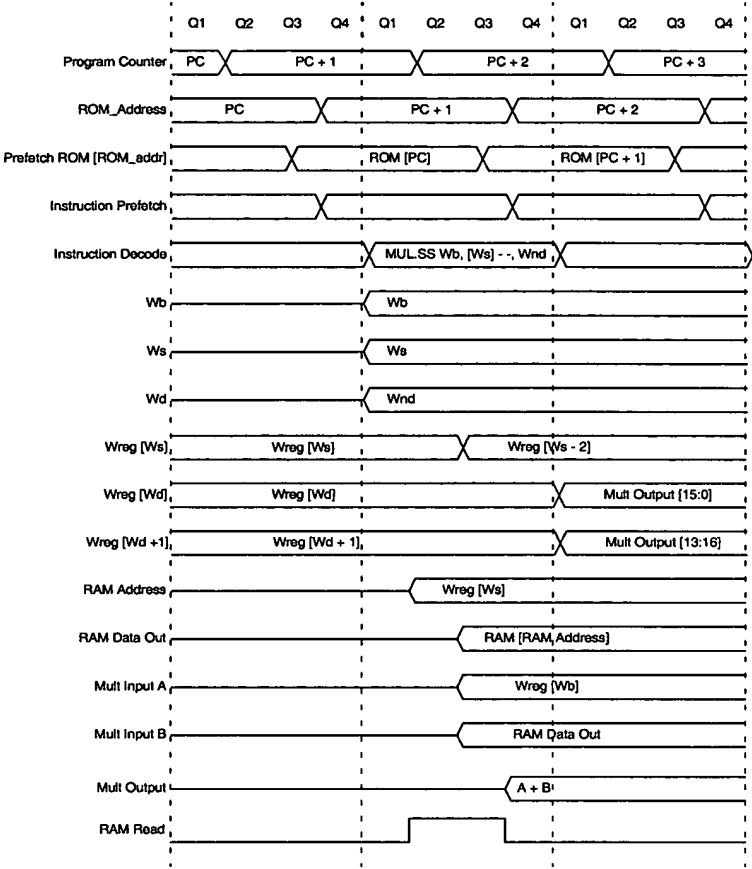


FIGURE 0-29: FLOW DIAGRAM MULWF

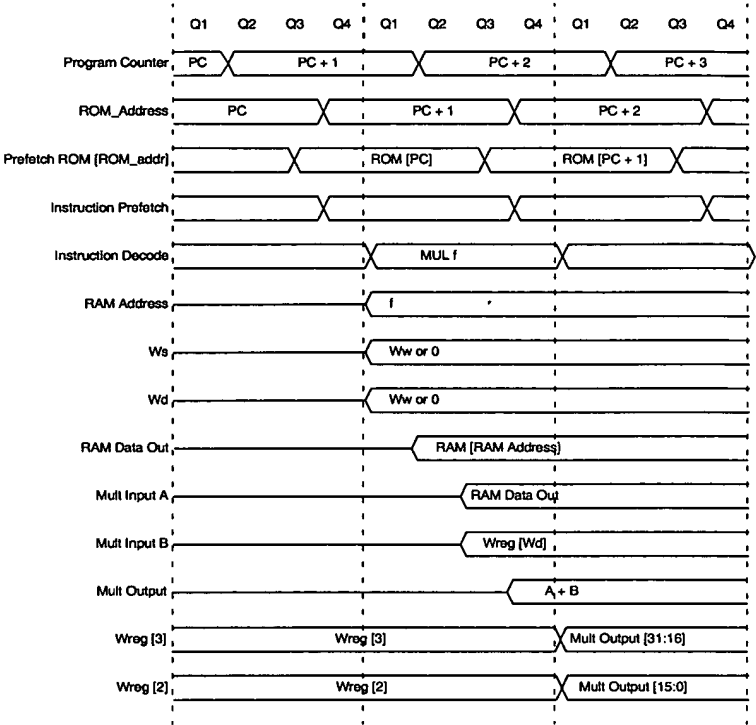


FIGURE 0-30: FLOW DIAGRAM ALL BRANCHES

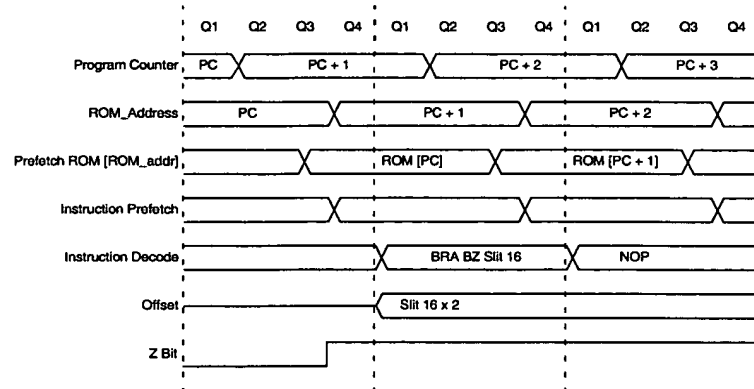


FIGURE 0-31: FLOW DIAGRAM BRAW

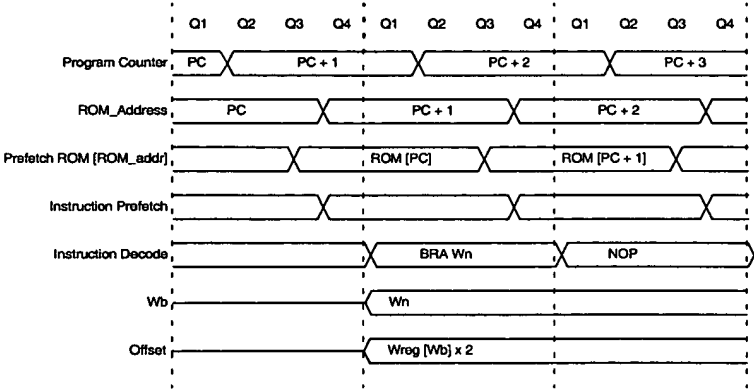
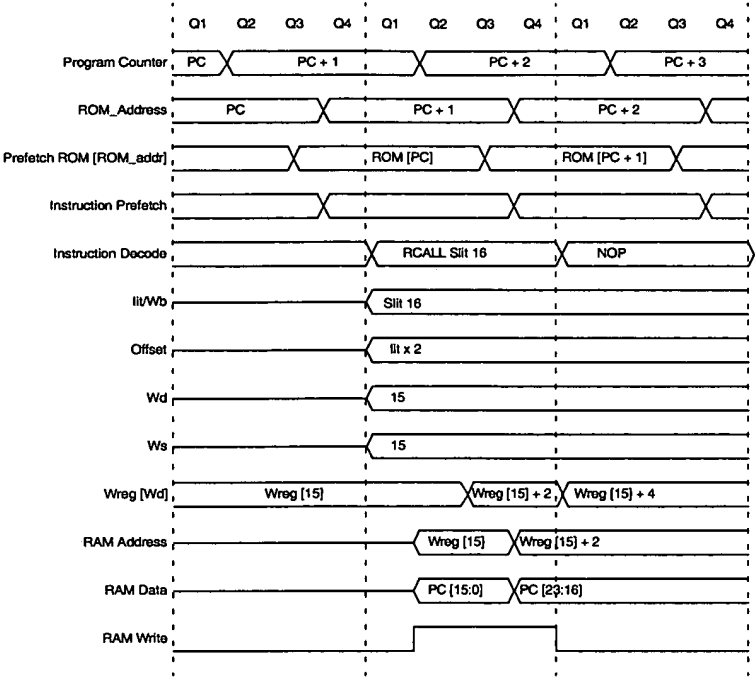


FIGURE 0-32: FLOW DIAGRAM RCALL, RCALLW



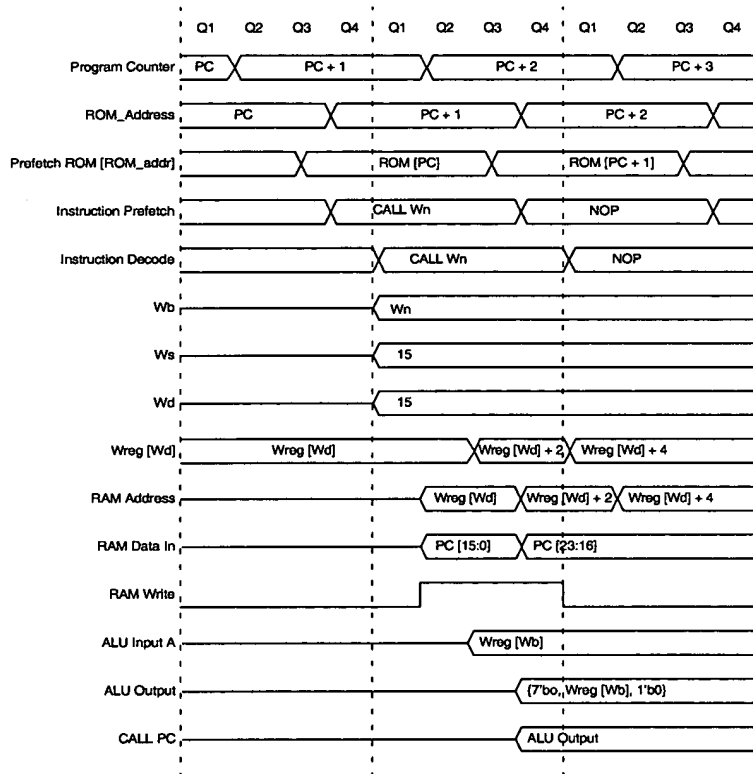
[illegible]

FIGURE 0-34: FLOW DIAGRAM CALL

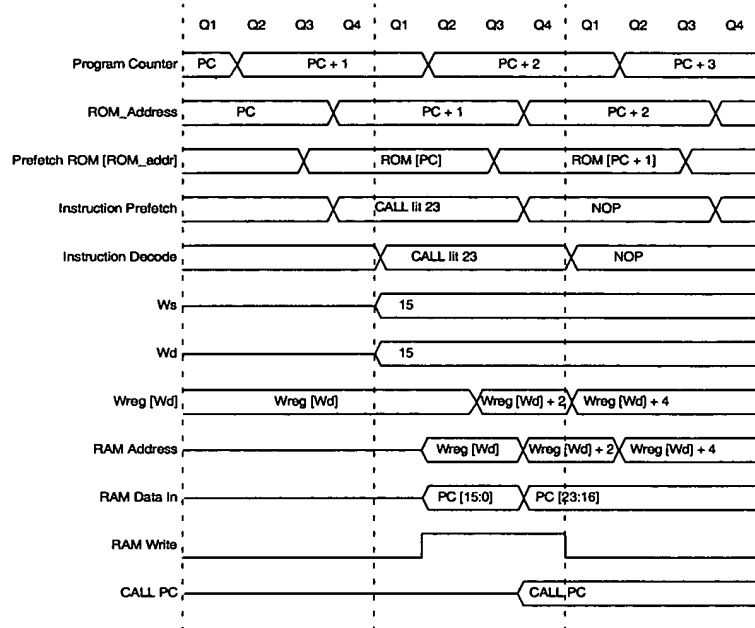




FIGURE 0-35: FLOW DIAGRAM GOTOW

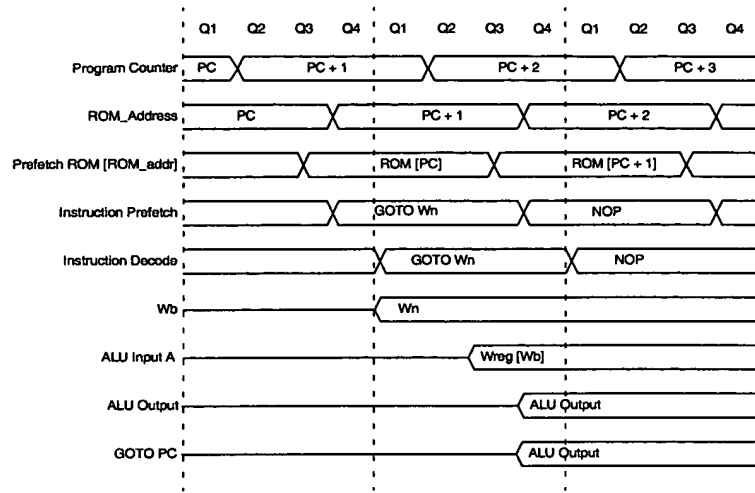
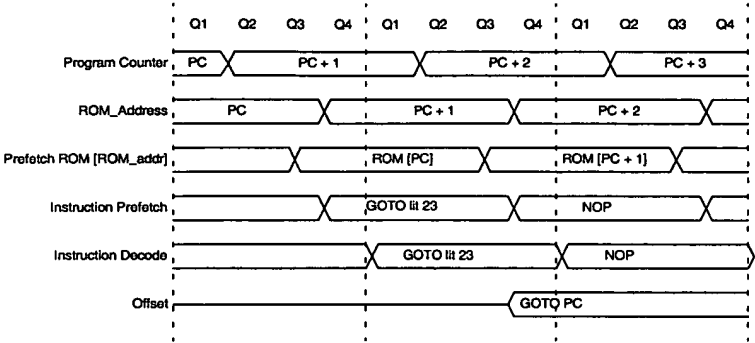


FIGURE 0-36: FLOW DIAGRAM GOTO



09870457 060101

FIGURE 0-37: FLOW DIAGRAM LNK

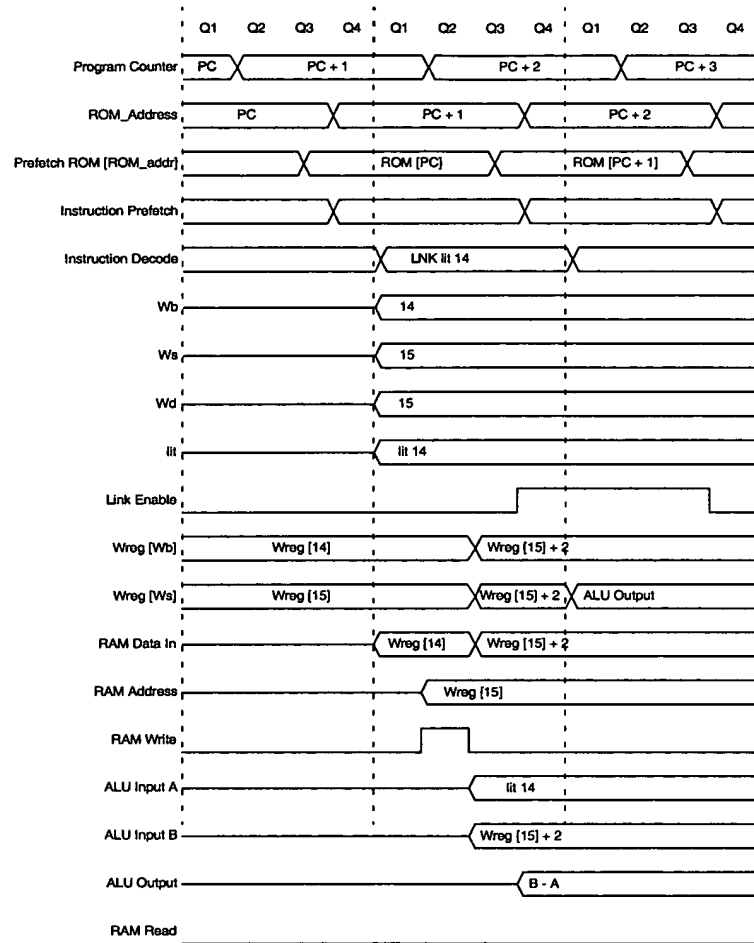


FIGURE 0-38: FLOW DIAGRAM ULNK

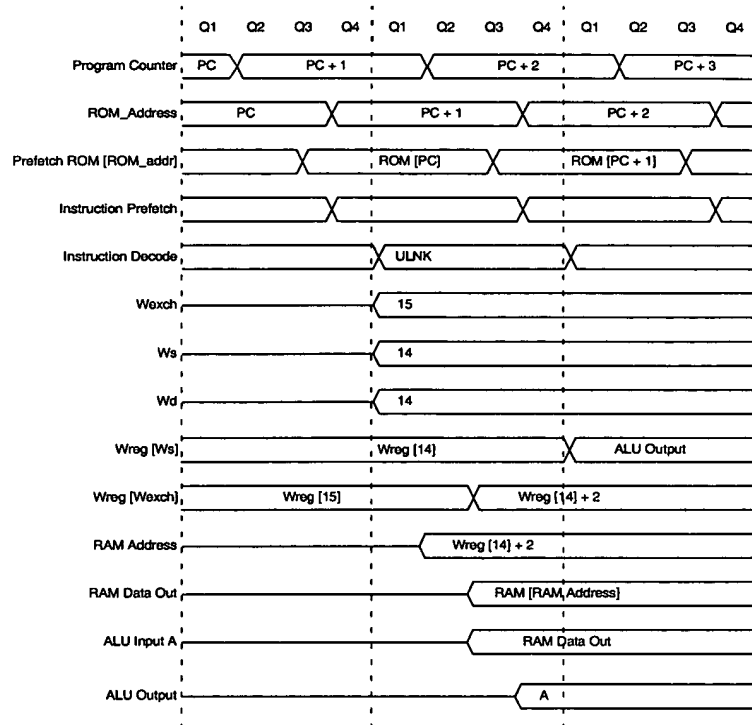


FIGURE 0-39: FLOW DIAGRAM DAW

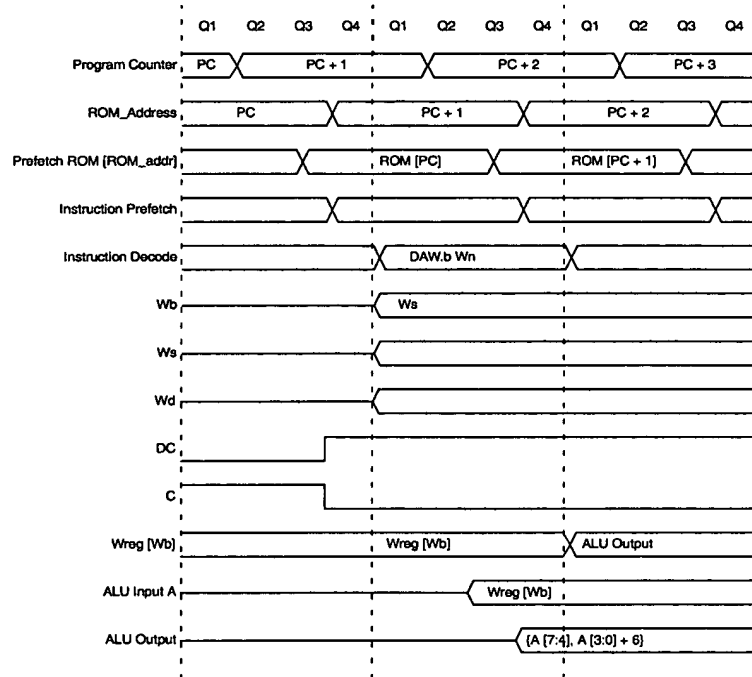


FIGURE 0-40: FLOW DIAGRAM SCRATCH

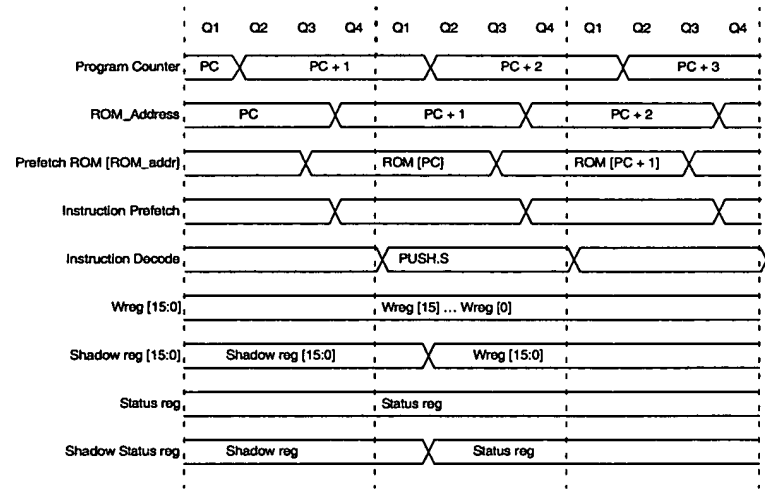


FIGURE 0-41: FLOW DIAGRAM ITCH

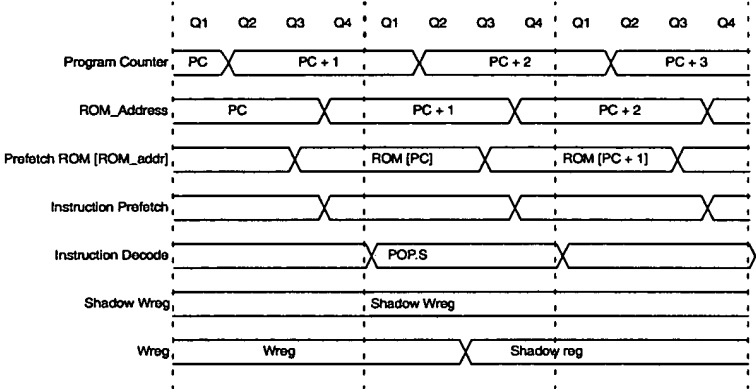


FIGURE 0-42: FLOW DIAGRAM PUSH

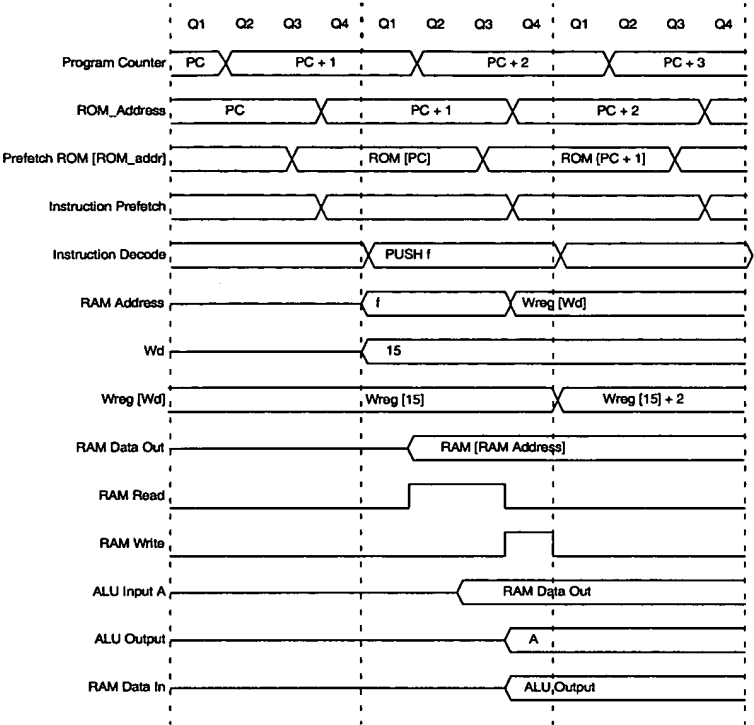




FIGURE 0-43: FLOW DIAGRAM POP

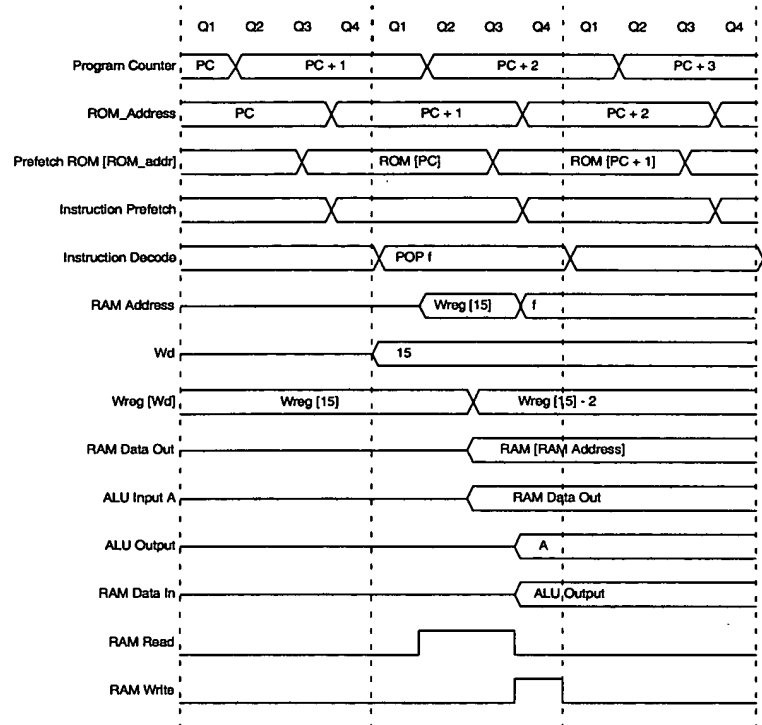
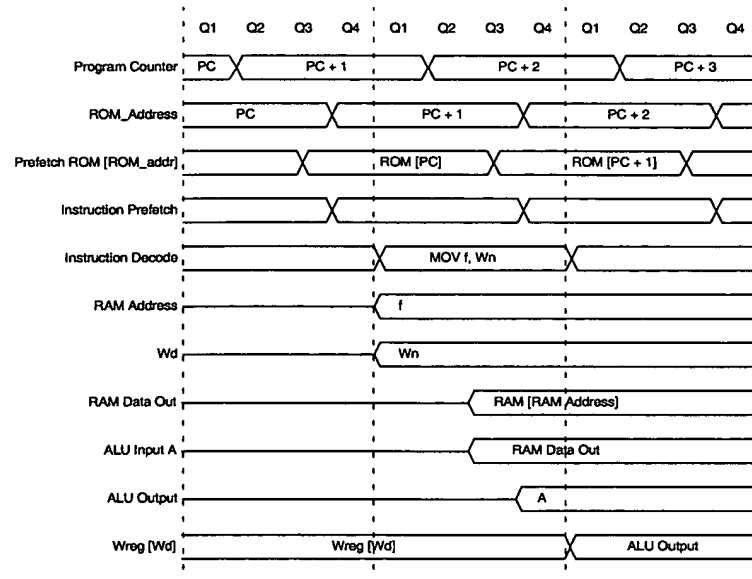
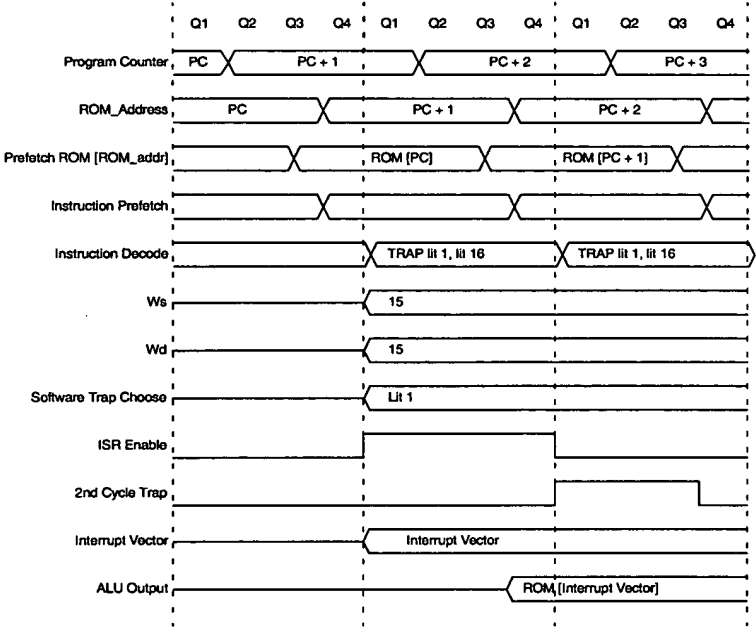


FIGURE 0-44: FLOW DIAGRAM LDW



00000000000000000000000000000000

FIGURE 0-45: FLOW DIAGRAM TRAP



The diagram illustrates the timing of various signals during the execution of an instruction. The clock is divided into 12 cycles, grouped by vertical dashed lines. The signals are as follows:

- Program Counter:** Holds PC in cycle 1, PC+1 in cycles 2-3, PC+2 in cycles 4-5, and PC+3 in cycles 6-7.
- ROM\_Address:** Holds PC in cycle 1, PC+1 in cycles 2-3, and PC+2 in cycles 4-5.
- Prefetch ROM [ROM\_addr]:** Holds ROM[PC] in cycles 2-3 and ROM[PC+1] in cycles 4-5.
- Instruction Prefetch:** Active (high) in cycles 2-3 and 4-5.
- Instruction Decode:** Active (high) in cycles 4-5, labeled as DISI bit 14.
- Number of DISI Count:** Holds bit 14 - 1 in cycles 6-7 and bit 14 - 2 in cycles 8-9.
- Interrupt Disable Signal:** Active (high) from cycle 6 to the end of the diagram.

FIGURE 0-47: FLOW DIAGRAM LDW

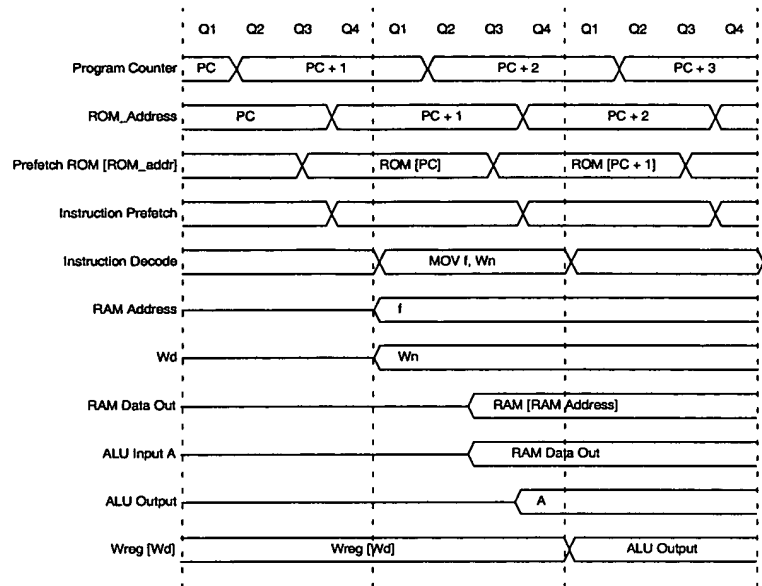


FIGURE 0-48: FLOW DIAGRAM DO, DOW

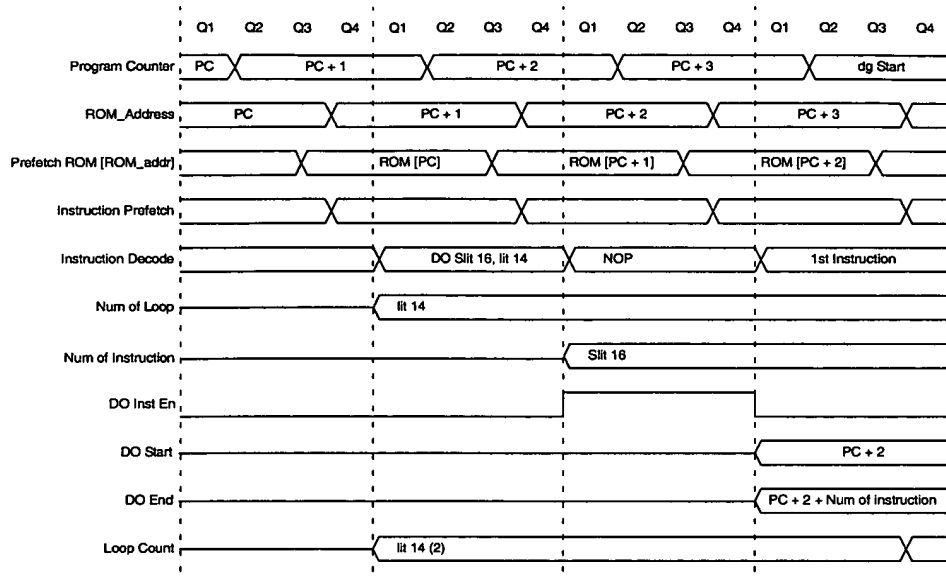
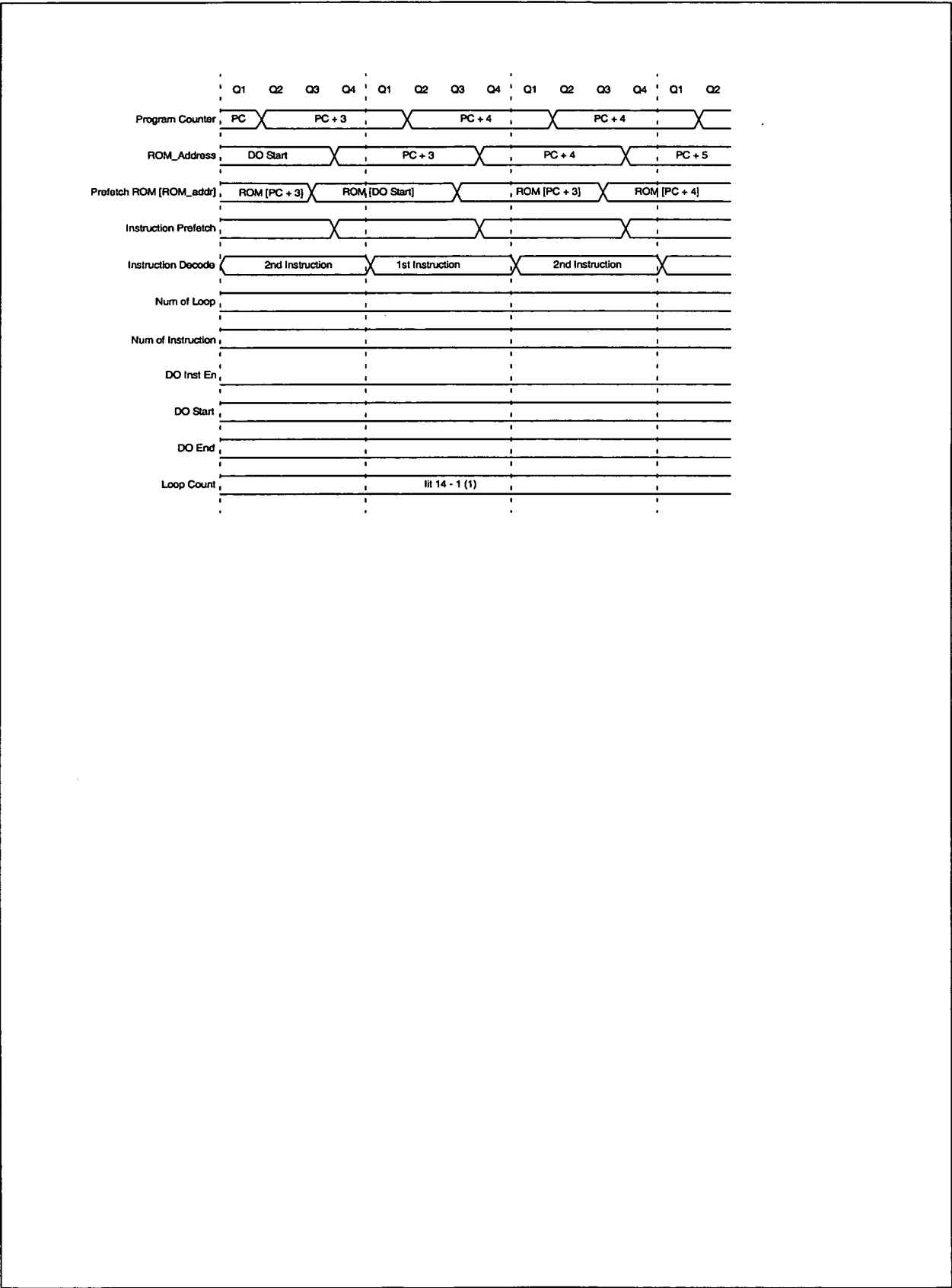


FIGURE 0-49: FLOW DIAGRAM DO, DOW CONT



09870457-060101

FIGURE 0-50: FLOW DIAGRAM MAC, CLRAC, EDAC, SQRAC, MOVSAC

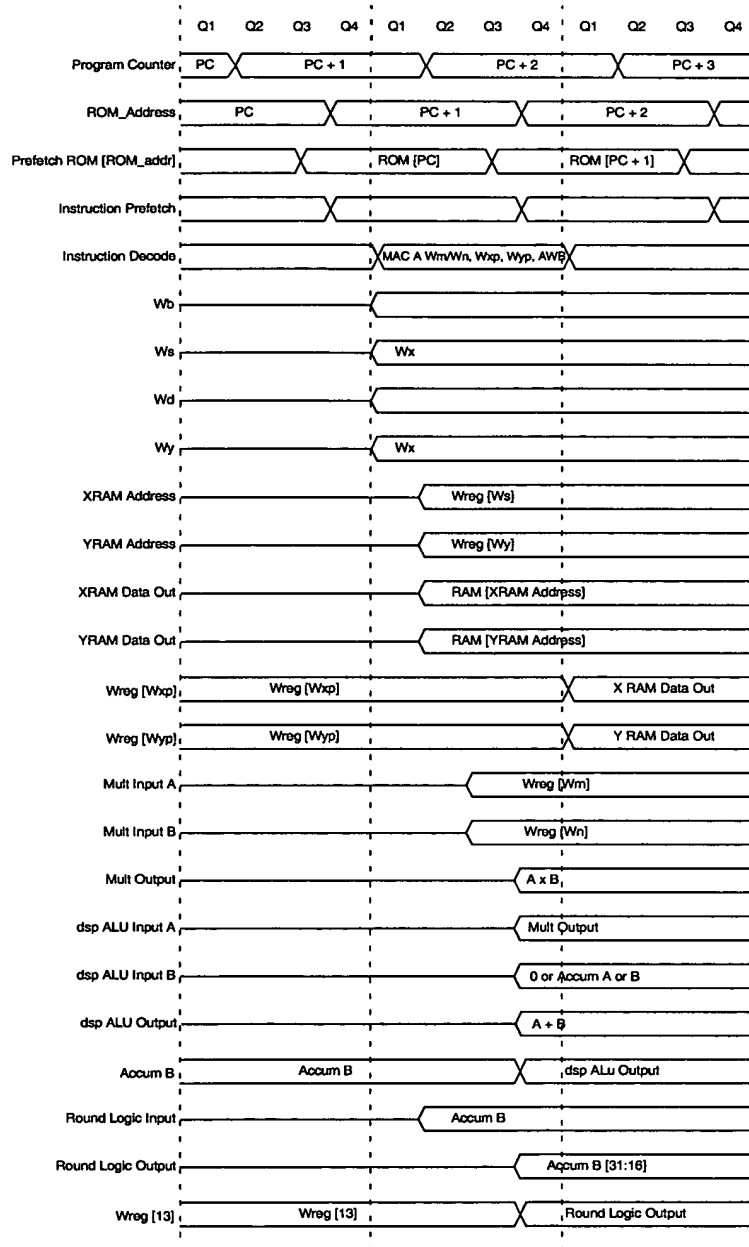
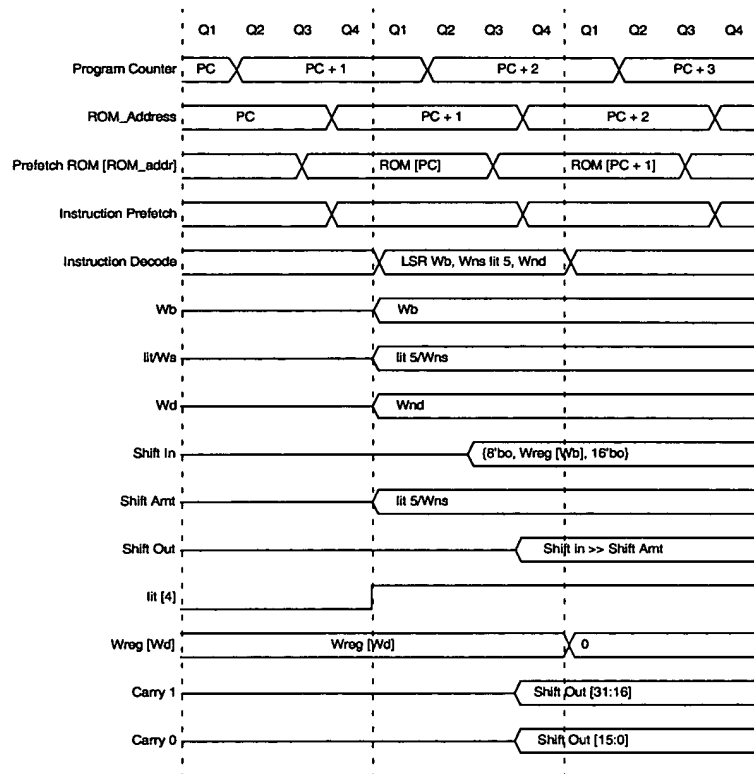




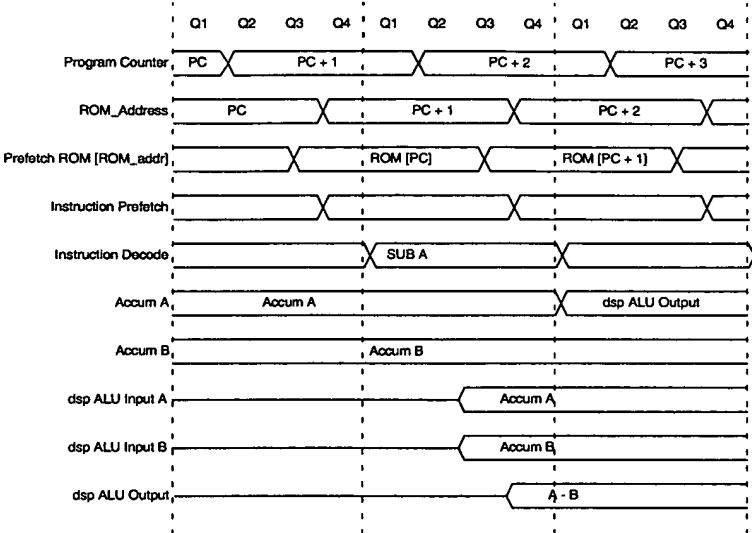


FIGURE 0-52: FLOW DIAGRAM LSRW, LSRK, ASRK, ASRW, SLW, SLK



05870457-060101

FIGURE 0-53: FLOW DIAGRAM ADDAB, NEGAB, SUBAB



09870457-060101

FIGURE 0-54: FLOW DIAGRAM ADDAC

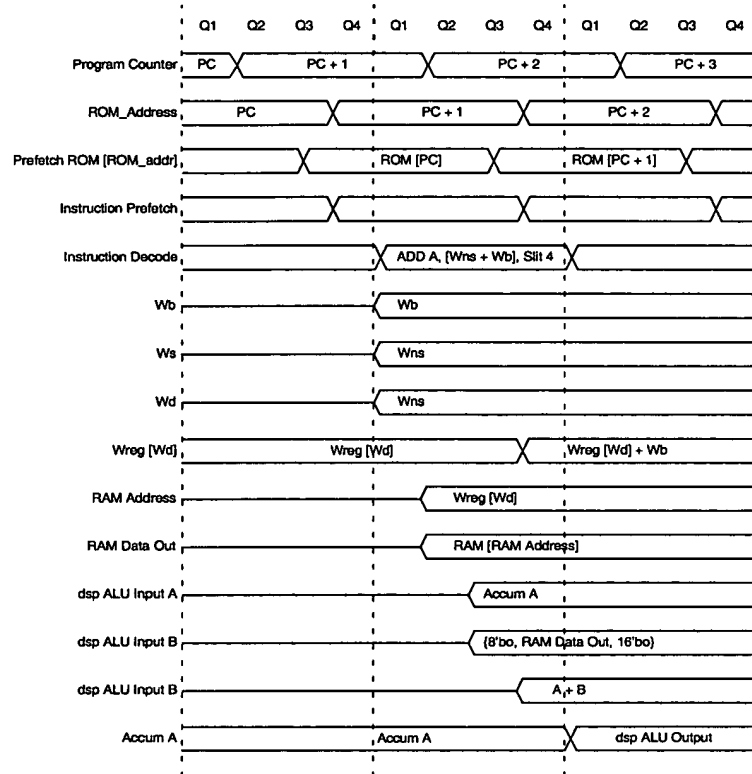
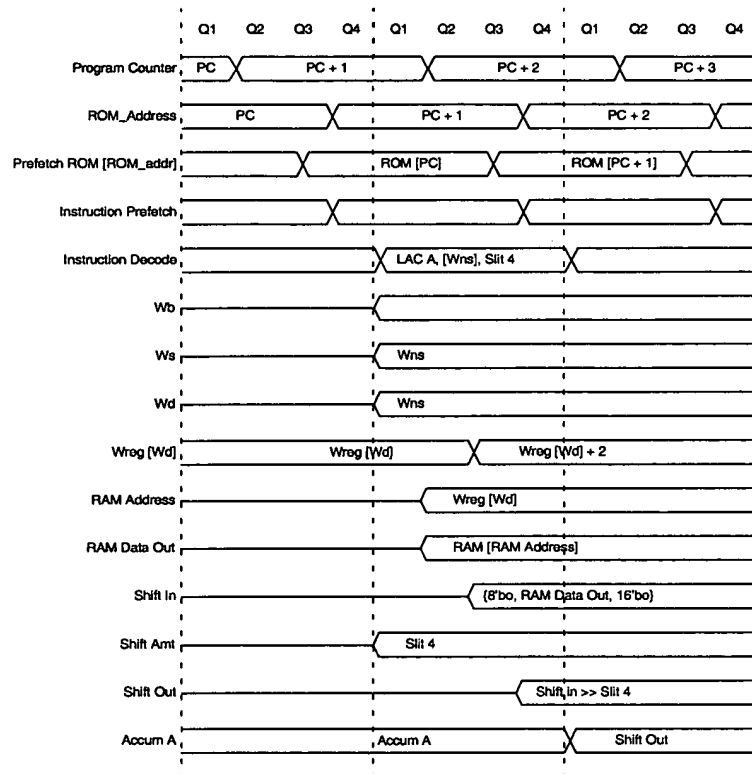


FIGURE 0-55: FLOW DIAGRAM LAC



The diagram illustrates the timing of the instruction `SAC.R A, [Wnd] ++, Slt 4`. The execution is divided into four 4-cycle blocks (Q1-Q4). The Program Counter (PC) increments by 1 every 4 cycles. The ROM Address is updated every 4 cycles. The Prefetch ROM signal is active during the first 4 cycles of each block. The Instruction Prefetch signal is active during the first 4 cycles of each block. The Instruction Decode signal is active during the first 4 cycles of each block. The Wb, Ws, and Wd signals are active during the first 4 cycles of each block. The Wreg [Wd] signal is active during the first 4 cycles of each block. The RAM Address signal is active during the first 4 cycles of each block. The Accum A signal is active during the first 4 cycles of each block. The Shift Amt signal is active during the first 4 cycles of each block. The Shift Input signal is active during the first 4 cycles of each block. The Shift Output signal is active during the first 4 cycles of each block. The Round In signal is active during the first 4 cycles of each block. The Round Out signal is active during the first 4 cycles of each block. The RAM Data In signal is active during the first 4 cycles of each block. The RAM Write signal is active during the first 4 cycles of each block.

[illegible]

FIGURE 0-57: FLOW DIAGRAM SFTACK, SFTAC

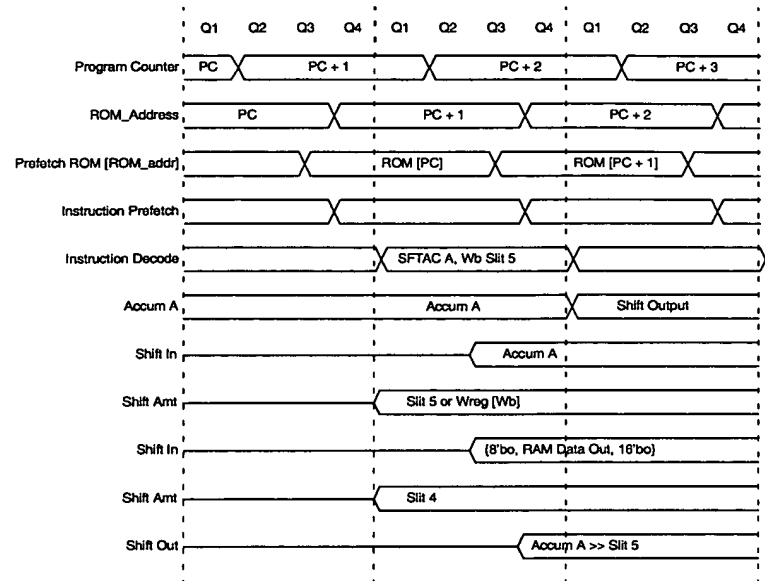
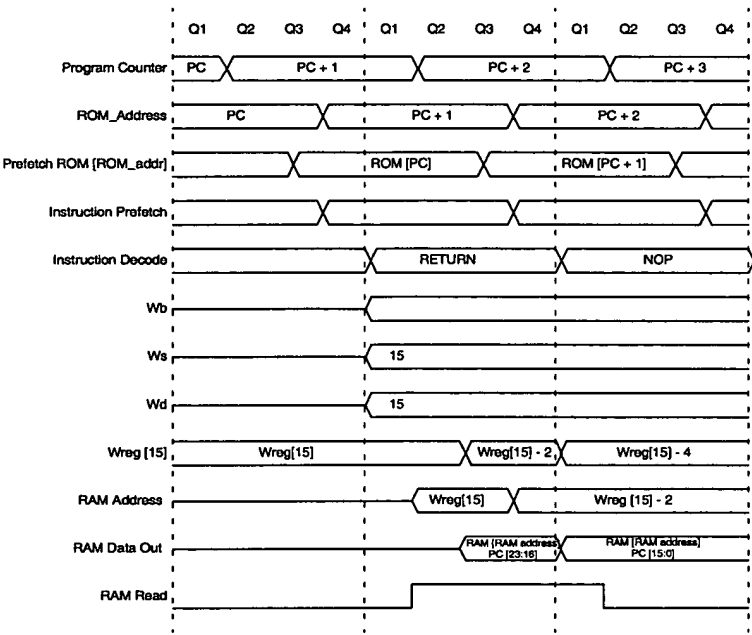


FIGURE 0-58: FLOW DIAGRAM RETURN, RE, TFIE



09370457-03010



FIGURE 0-59: FLOW DIAGRAM MSLK, MSRK, MSLW, MSRW

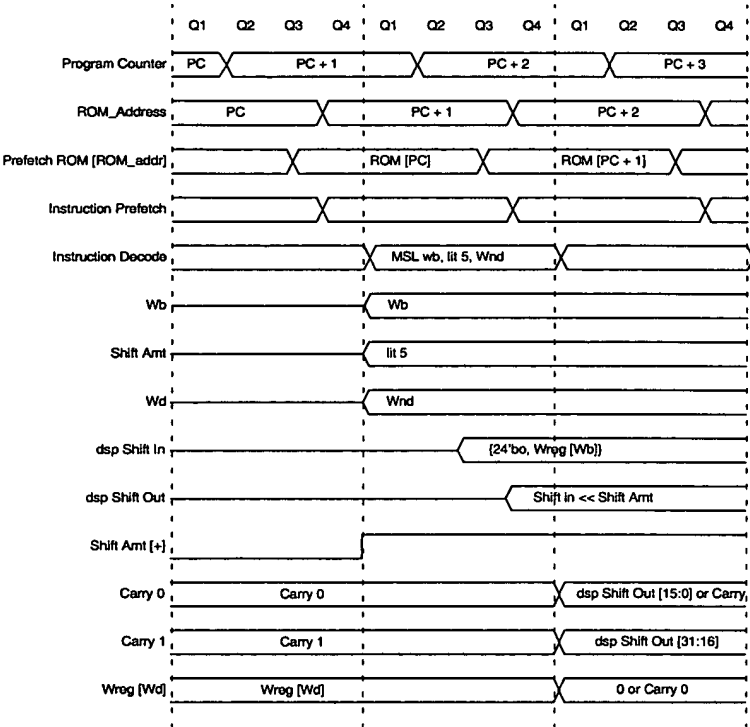


FIGURE 0-60: FLOW DIAGRAM FBCL, FBCR, FFOL, FFOR, FFIL, FFIR

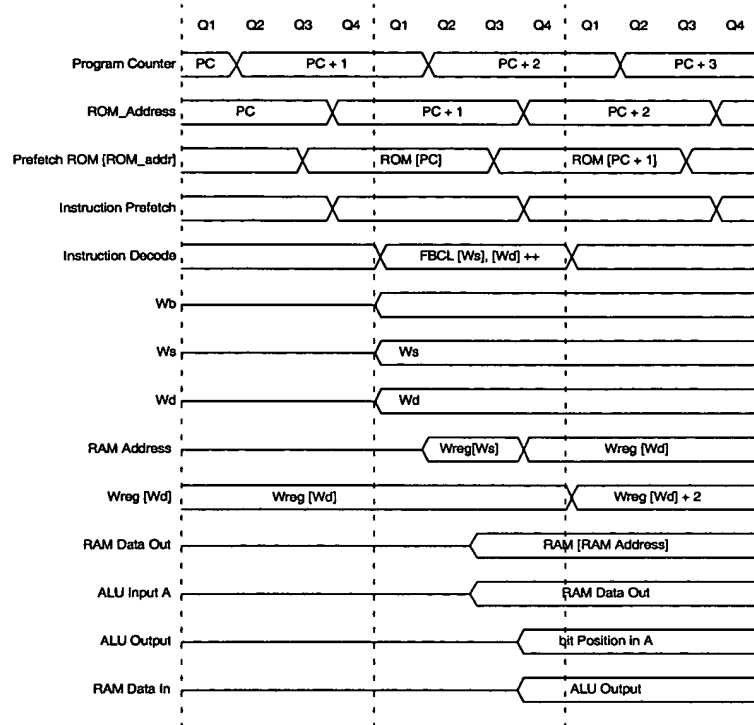
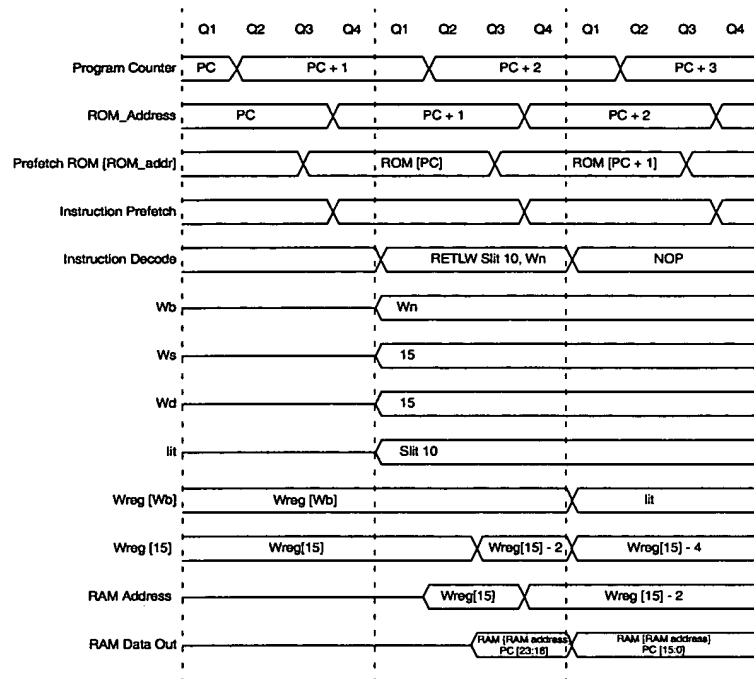


FIGURE 0-61: FLOW DIAGRAM RETLW



09870457-060101

FIGURE 0-62: FLOW DIAGRAM REPEAT, REPEAT W

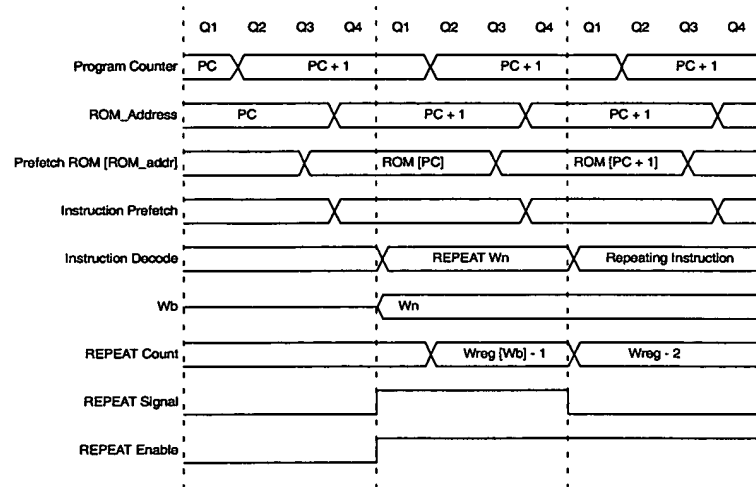
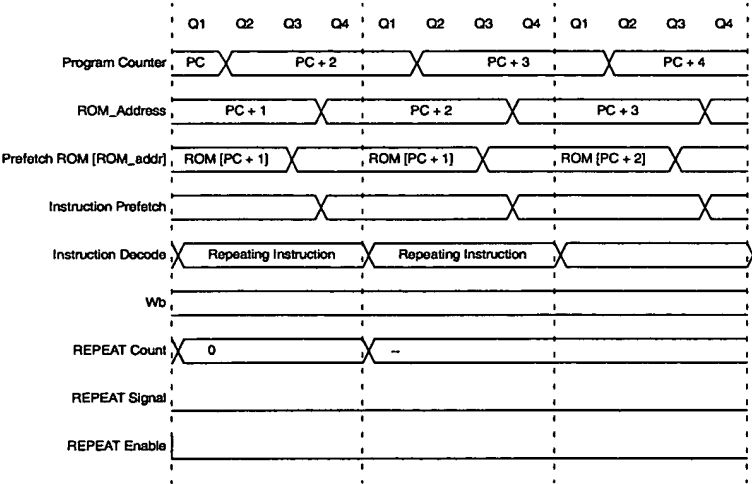


FIGURE 0-63: FLOW DIAGRAM REPEAT, REPEAT W (CONTD)



APPENDIX D

09870457.060101  
FOR 090 / 510 / 860

## 1.0 ARCHITECTURAL DESCRIPTION

The Roadrunner core is a 16-bit (data) modified Harvard architecture with a greatly enhanced 'C18-like' instruction set including significant support for DSP.

### 1.1 Core Overview

The core has a 24-bit instruction word, with a variable length opcode field. The PC is 24-bits wide (with the LS-bit always clear, see Section 1.3.1), addressing up to 8M long words (23-bits). An 'C18-like' instruction prefetch mechanism is used to help maintain throughput. Deeper levels of pipelining have been intentionally avoided to maintain good real-time performance. Unconditional overhead free program loop constructs are supported using the DO and REPEAT instructions, both of which are interruptable at any point.

The working register array has been extended to 16 x 16-bit registers, each of which can act as data, address or offset registers. One working register (W15) operates as a software stack for interrupts and calls.

The data space is 32K words of word or byte addressable space which is split into two blocks referred to as X and Y data memory. Each block has its own independent Address Generation Unit (AGU). Most instructions operate solely through the X memory AGU which will make it appear as one linear space encompassing all data space. The MAC class of DSP instructions will operate through both the X and Y AGUs, splitting the data address space into two parts (see Section 1.2.4). The X and Y data space boundary is arbitrary and defined through the address decode of each memory array.

The upper 32K bytes of data space memory can optionally be mapped into the lower half (user space) of program space at any 16K program word boundary defined by the 8-bit Data Space Program PAGE (DSP-PAG) register. This lets any instruction to access program space as if it were data space (other than the additional access cycle it consumes) plus allows external RAM hooked onto the external program space to be mapped into data space, effectively providing an external data space bus.

Overhead free circular buffers (modulo addressing) are supported in both X and Y address spaces. They are intended to remove the loop overhead for DSP algorithms but X modulo addressing can be universally applied using any instructions.

The X AGU also supports bit reverse addressing to greatly simplify input or output data reordering for radix-2 FFT algorithms.

The Instruction Set Architecture (ISA) has been significantly enhanced beyond that of the C18 but maintains an acceptable level of backward compatibility. All C18 instructions and addressing modes are supported either directly or through simple macros (see xxxx). Many of the ISA enhancements have been driven by compiler efficiency needs (see Section 1.1.1).

The core supports inherent (no operand), relative, literal, memory direct and 4 groups of addressing modes (MODE1, MODE2, MODE3 and MODE4) for register direct and register indirect modes. Each group offers up to 6 addressing modes. Instructions are associated with predefined addressing modes depending upon their functional requirements.

For most instructions, the core is capable of executing a data (or program data) memory read, a working register (data) read, a data memory write and a program (instruction) memory read per instruction cycle. As a result, 3 operand instructions can be supported, allowing  $A+B=C$  operations to be executed in a single cycle.

A DSP engine has been included to significantly enhance the core arithmetic capability and throughput. It features a high speed 16-bit by 16-bit multiplier, a 40-bit ALU, two 40-bit saturating accumulators and a 40-bit bidirectional barrel shifter. The barrel shifter is capable of shifting a 40-bit value up to 15 bits right or up to 16-bits left in a single cycle. The DSP instructions operate seamlessly with all other instructions and have been designed for optimal real-time performance. The MAC class of instructions can concurrently fetch two data operands from memory while multiplying two W registers. This requires that the data space be split for these instructions and linear for all others. This is achieved in a transparent and flexible manner through dedicating certain working registers to each address space for the MAC class of instructions.

The core features a vectored exception scheme with 15 individually prioritized vectors. The exceptions consist of reset, 7 traps and 8 interrupts. One interrupt level may be selected (typically the highest one) to execute as a fast (1 cycle entry, 1 cycle exit) interrupt. This function is actually an extension of the logic required to allow a REPEAT instruction loop to be interrupted which can significantly reduce latency in some application.

A block diagram of the core is shown in Figure 1-1.

#### 1.1.1 Compiler Driven Enhancements

In addition to DSP performance requirements, the core architecture was strongly influenced by recommendations which would lead to a more efficient (code size and speed) C compiler.

1. For most instructions, the core is capable of executing a data (or program data) memory read, a working register (data) read, a data memory write and a program (instruction) memory read

per instruction cycle. As a result, 3 operand instructions can be supported, allowing  $A+B=C$  operations to be executed in a single cycle.

2. Instruction addressing modes are significantly more flexible than those of the C18, and are matched closely to compiler needs.
3. The working register array has been extended to 16 x 16-bit registers, each of which can act as data, address or offset registers. One working register (W15) operates as a software stack for interrupts and calls.
4. Linear indirect access of all data space is possible, plus the memory direct address range has been extended to 8Kbytes (256bytes in C18). This together with the addition of 16-bit direct address LOAD and STORE instructions has allowed the C18 data space memory banking scheme to be eliminated.
5. Linear indirect access of 32K word (64K byte) pages within program space (user and test space) is possible using any working register via new table read and write instructions.
6. Part of data space can be mapped into program space, allowing constant data to be accessed as if it were in data space.

#### 1.1.2 Instruction Fetch Mechanism

The core does not support an instruction pipeline. A pre-fetching mechanism accesses instruction a cycle ahead to maximize available execution time. Most instructions execute in a single cycle. Exceptions are:

1. Flow control instructions and interrupts where the ISR (instruction register) and pre-fetch buffer must be flushed and refilled.
2. Instructions where one operand is to be fetched from program space (using any method). These operations consume 2 cycles (with the notable exception of the MAC class of DSP instructions executed within a REPEAT loop which executes in 1 cycle).

Most instructions access data as required during instruction execution. Instructions which utilize the multiplier array must have data available at the beginning of the instruction cycle. Consequently, this data must be prefetched, usually by the preceding instruction, resulting in a simple out of order data processing model.



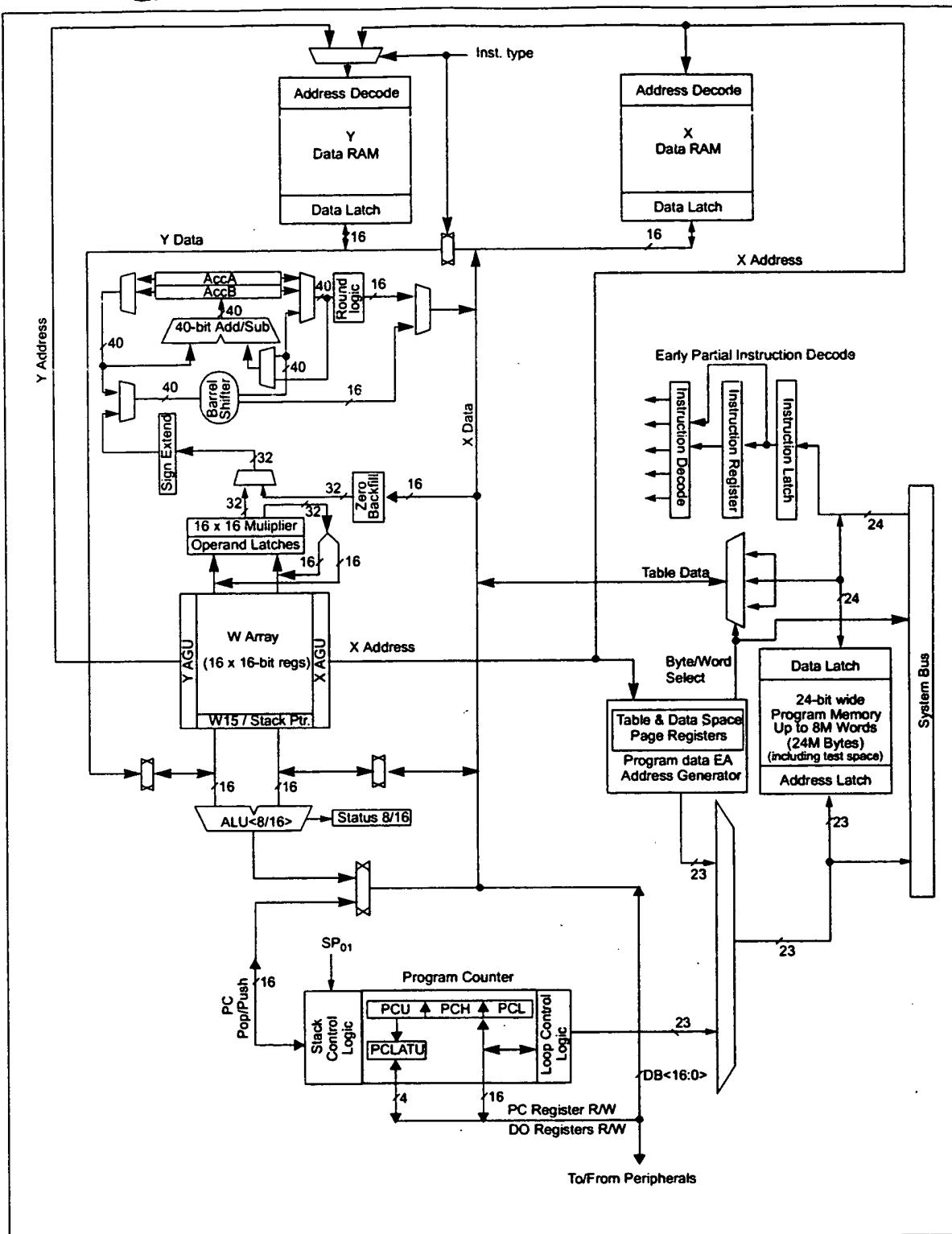


FIGURE 1-1: ROADRUNNER CPU CORE BLOCK DIAGRAM

## 1.2 Data Address Space

The core features one program space and two data spaces. The data spaces can be considered either separately (for some DSP instructions) or together as one linear address range (for MCU instructions). The data spaces are accessed using two Address Generation Units (AGUs) and separate data paths.

### 1.2.1 Data Spaces

The X AGU is used by all instructions and supports all addressing modes. It also supports modulo and bit reversed addressing for any instructions subject to addressing mode restrictions (see Section 4.2.2). The X data path is the return data path for all single data space access instructions.

The Y AGU and data path are used in concert with the X AGU by the MAC class of instructions to provide two concurrent data read paths. No writes occur across the Y-bus. This class of instructions dedicate two W register pointers, W6 and W7, to always operate through the Y AGU and address Y data space independently from X data space. Note that during accumulator write-back, the data address space is considered combined X and Y, so the write will occur across the X-bus. Consequently, it can be to any address irrespective of where the EA is directed.

The Y AGU only supports MODE4 post modification addressing modes (see Section 4.1.4) associated with the MAC class of instructions. It also supports modulo addressing for automated circular buffers. Of course, all other instructions can access the Y data address space through the X AGU when it is regarded as part of the composite linear space.

The boundary between the X and Y data spaces is arbitrary and is defined by the memory address decode only (the CPU has no knowledge of the physical location of X or Y memory). The boundary is not user programmable but may change from variant to variant. Obviously, to present a linear data space to the MCU instructions, the address spaces of X and Y data spaces must be contiguous but this is not an architectural necessity. Note that any memory located between 0x8000 and 0xFFFF will not be accessible when program space visibility is enabled for this address space.

**Note:** As address space 0x8000 to 0xFFFF can map to a single memory in program space, it must be assigned to either X or Y space (but not both since concurrent accesses from the same space are not possible).

All (effective addresses) are 16-bits wide and point to bytes within the data space to facilitate backward compatibility with the C18. Consequently, the data space address range is 64K bytes or 32K words.

### 1.2.2 Data Space Width

The core data width is 16-bits. All internal registers and data space memory are organized as 16-bits wide (some CPU registers are not 16-bits wide - refer to Figure 1-33). Data space memory is organized in byte addressable, 16-bit wide blocks. Byte addressability requires independent byte write signals for upper and lower bytes.

### 1.2.3 Data Alignment

To help maintain C18 backward compatibility and improve data space memory usage efficiency, the ISA supports both word and byte operations. Data is aligned in data memory and registers as words, but all data space EAs resolve to bytes. Data byte reads will read the complete word which contains the byte, using the LS-bit of any EA to determine which byte to select. The selected byte is placed onto the LS-byte of the X data path (no byte accesses are possible from the Y data path as the MAC class of instruction can only fetch words). That is, data memory and registers are organized as two parallel byte wide entities with shared (word) address decode but separate write lines. Data byte writes will only write to the corresponding side of the array or register which matches the byte address. For word accesses, the LS-bit of the EA is ignored (don't care).

**Note:** Byte reads will always read the entire word, so mechanisms to clear or set peripheral status bits when read (e.g. quick flag clearing mechanisms) are not allowed.

As a consequence of this byte accessibility, all effective address calculations (including those generated by the DSP operations which are restricted to word size) must be scaled to step through word aligned memory. For example, the core must recognize that post modified register indirect addressing mode,  $[Ws] += 1$ , will result in a value of  $Ws + 1$  for byte operations and  $Ws + 2$  for word operations.

All word accesses must be aligned (to an even address). Mis-aligned word data fetches are not supported so care must therefore be taken when mixing byte and word operations or translating from C18 code. Should a mis-aligned read or write be attempted, an address fault trap will be forced. Depending upon where the fault occurred in the instruction cycle, the Q1/Q2 access (typically a read) and/or the Q3/Q4 access (typically a write) for the instruction underway will be inhibited, and the PC will not be incremented. The trap will then be taken, allowing the system and/or user to examine the machine state prior to execution of the address fault.

	15	MS byte	8	7	LS byte	0	
0001	Byte1				Byte 0		0000
0003	Byte3				Byte 2		0002
0005	Byte5				Byte 4		0004

**FIGURE 1-2: DATA ALIGNMENT**

All byte loads into any W register are loaded into the LS-byte. The MS-byte is not modified.

**Note:** Byte operations use the 16-bit ALU and can produce results in excess of 8-bits. However, to maintain C18 backwards compatibility, the ALU result from all byte operations is written back as a byte (i.e. MS byte not modified), and the status register is updated based only upon the state of the LS-byte of the result.

A sign extend (SE) instruction is provided to allow users to translate 8-bit signed data to 16-bit signed values. Alternatively, for 16-bit unsigned data, users can clear the MS-byte of any W register though executing a CLR.b instruction on the appropriate address.

**Note:** Care must be taken when mixing byte and word size instructions/operands.

Although most instructions are capable of operating on word or byte data sizes, it should be noted that the DSP and some other new instructions operate on words only.

#### 1.2.4 Data Space Memory Map

The data space memory is split into two blocks, X and Y data space. A key element of this architecture is that Y space is a subset of X space, and is fully contained within X space. In order to provide an apparent linear addressing space, X and Y space would typically have contiguous addresses (though this is not an architectural necessity).

When executing any instruction other than a MAC class one, the X block consists of the entire 64Kbyte data address space (including all Y addresses). When executing a MAC class of instruction, the X block consists of the entire 64Kbyte data address space less the Y address block for data reads (only). In other words, the full address space is available to all instructions other than the MAC class. During Q1/Q2 data reads, the MAC class of instructions extracts the Y address space from data space and addresses it using EA's sourced from W6 and W7. The remaining data space is referred to as X space but could more accurately be described as "X-Y" space, and is concurrently addressed using W4 and W5 during the same Q1/Q2

data read portion of the cycle. Both "X-Y" and Y address spaces are concurrently accessed only by the MAC class of instruction.

Note that it is the register number (and instruction class) that determine which address space is accessed for data reads and not the EA. Consequently, the data space partitioning of Y address space is arbitrary. In all cases, should an EA point to unoccupied space, all zeros will be returned. For example, although Y address space is visible by all non-MAC class instructions using any addressing mode, an attempt by a MAC instruction to fetch data from that space using W4 or W5 (X space pointers) will return 0x0000.

An example data space memory map is shown in Figure 1-3. Note again that the partition between each address space is arbitrary and determined by the memory decode. Both X and Y address generation units (AGUs) can generate any effective address (EA) within a 64Kbyte range, however, EAs outwith the physical memory provided will return all zeros.

An 8Kbyte access space is reserved in X address memory space between 0x0000 and 0x1FFF which is directly addressable via a 13-bit absolute address field within all memory direct instructions. The remaining X address space and all of the Y address space is addressable indirectly. The whole of X data space is additionally addressable using LDW and STW instructions which support memory direct addressing with a 16-bit address field.

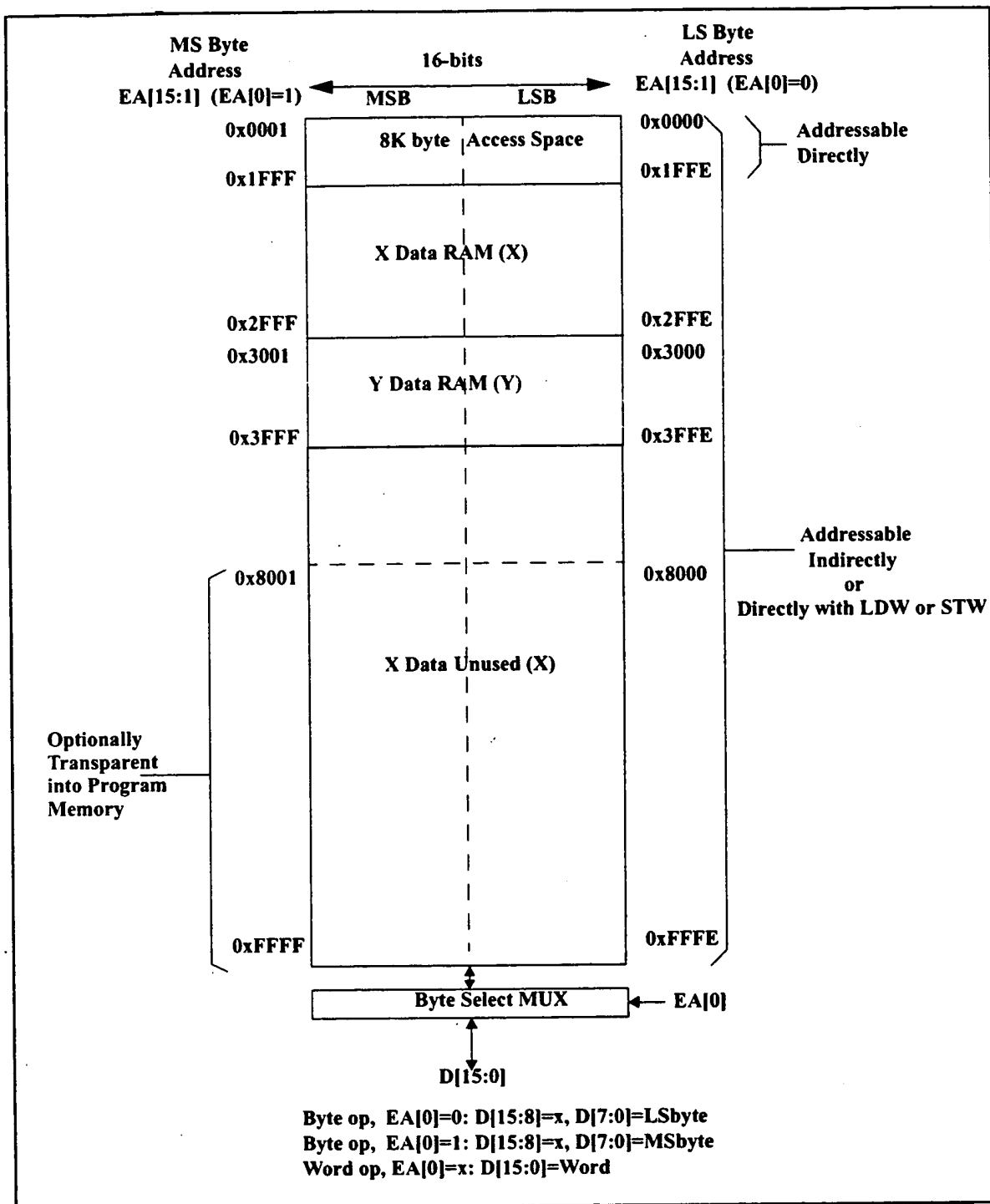
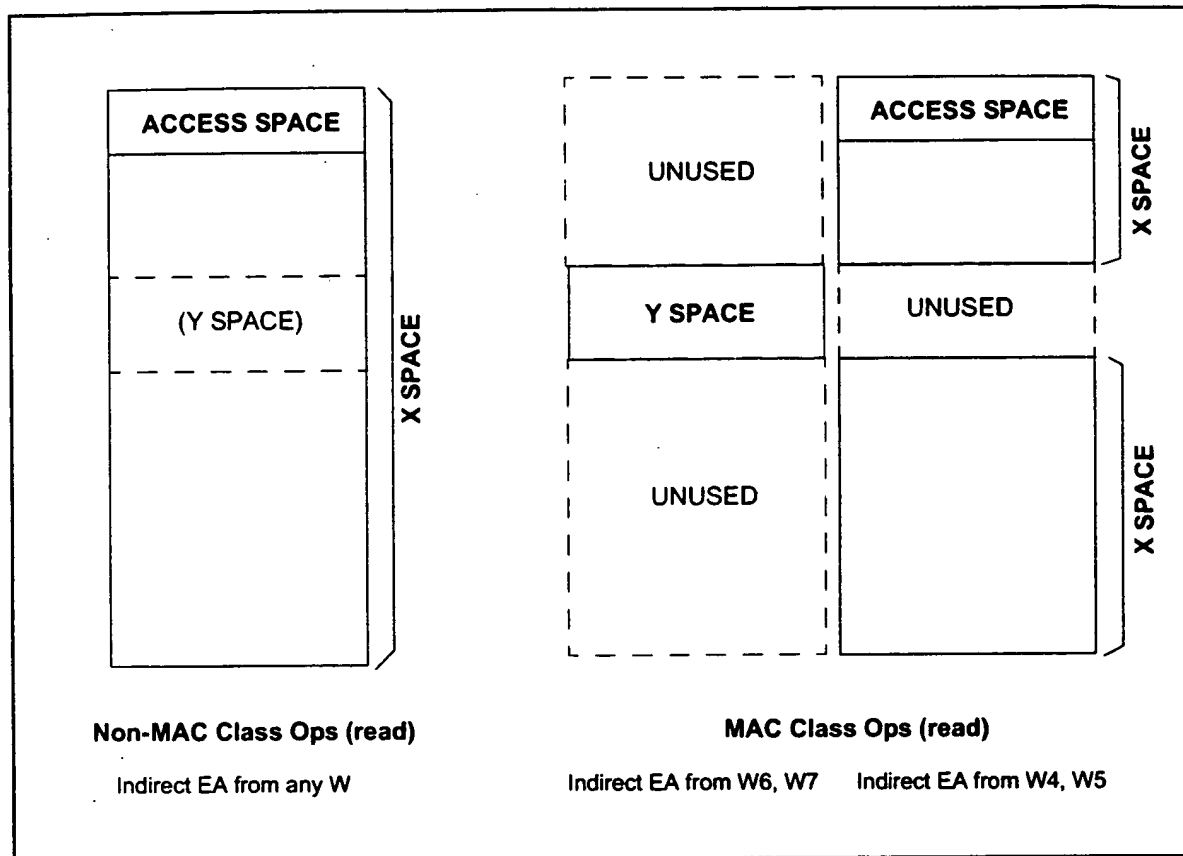


FIGURE 1-3: DATA SPACE MEMORY MAP EXAMPLE



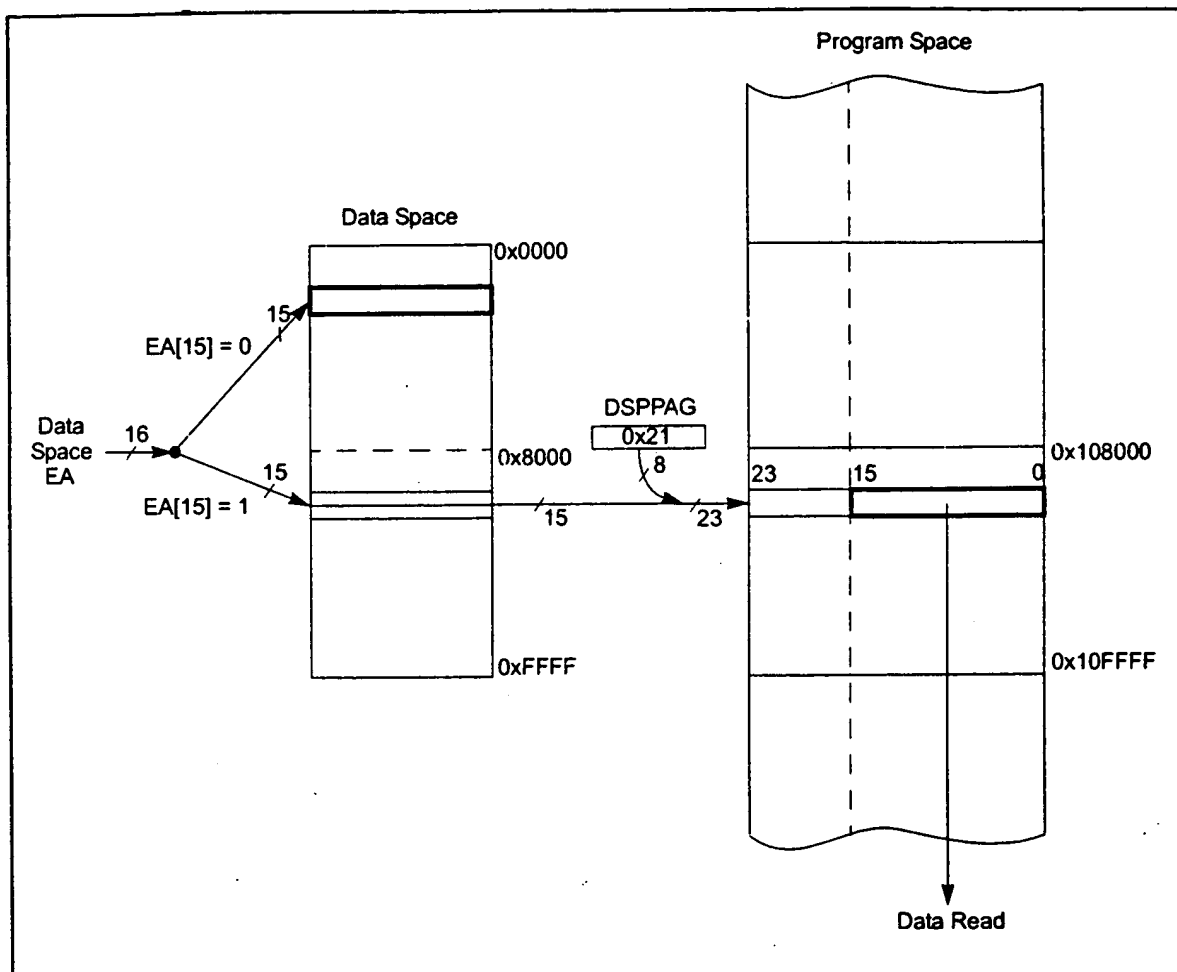
**FIGURE 1-4: DATA SPACE FOR MCU AND DSP (MAC CLASS) INSTRUCTIONS EXAMPLE**

### 1.2.5 Program Space Visibility from Data Space

The upper 32Kbytes of data space may optionally be mapped into any 16Kword program space page. This provides transparent access of stored constant data from X data space without the need to use special instructions (i.e. TBLRD, TBLWT instructions).

**Note:** Granularity of program space window may change, subject to conclusions of code security analysis.

This feature also allows the user to map the upper half of data space into an unused area of program memory and thus to the external bus (all unused internal addresses will be mapped externally). Through the placement of an external RAM at this address, external data space support is also provided. Data read and writes must therefore be supported to this address space. The effect of data writes to internal program space is defined in the Program Memory DOS-00204. Note that the external address map is now essentially no longer strictly Harvard as program and data memory are combined.



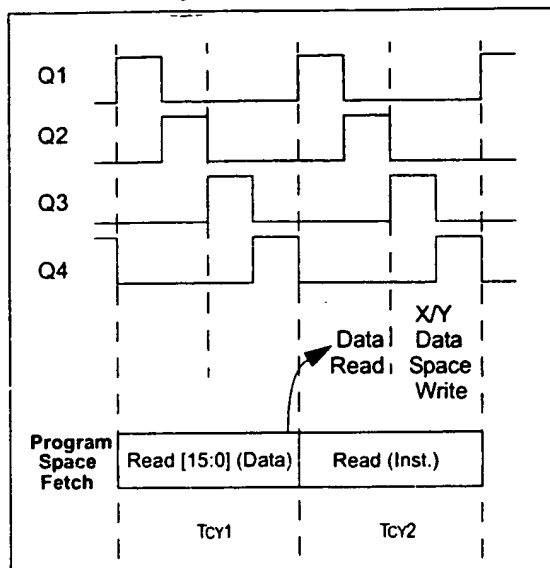
**FIGURE 1-5: DATA SPACE WINDOW INTO PROGRAM SPACE OPERATION**

Program space access through the data space occurs if the MS-bit of the data space EA is set and program space visibility is enabled by setting the PSV bit in the CORE Control register, CORCON. Most of the CORCON function relate to DSP operation so it is discussed in Section 2.0, DSP Engine.

**Note:** Depending upon FLASH setup & access time, the instruction may need to be at least partially pre-decoded during Q4 of the prior instruction. Evenso, this will remain a critical path, as the source EA cannot be evaluated until the data write completes in the prior instruction.

Data accesses to this area will add an addition cycle to the instruction being executed since two program memory fetches will be required. The data is fetched in the first cycle, which, other than for some instruction decode, is essentially a NOP. The next instruction is

prefetched in the second cycle while the current instruction completes execution (i.e. normal operation) as shown in Figure 1-6.



**FIGURE 1-6: PS DATA READ THROUGH DS**

Furthermore, instructions executing from internal program memory but accessing external data memory RAM will sustain additional delay due to wait state insertion. Read-modify-write operations will sustain twice the delay.

**Note:** The External Bus Interface (EBI) definition is not complete at this time, however, it is expected that the device will be required to insert an even number of Q clocks into the instruction cycle between Q2 and Q3, and between Q4 and Q1 (of the next cycle) for external data space accesses.

Although not an architectural necessity, a typical data space configuration would define Y data space to be outside this re-mappable area, making the visible program space map to X data space. Y data space will typically contain state (variable) data for DSP operations, and must therefore be RAM. X data space will typically contain coefficient (constant) data which could be NVM or initialized RAM.

Although each transparent data space address will map directly into a program address (see Figure 1-8), only the lower 16-bits of the 24-bit program word are used to contain the data. The upper 8-bits should be programmed to force an illegal instruction or software trap to maintain machine robustness.

For external accesses, data space would only require a 16-bit data path, with the trap instruction being automatically concatenated onto any 16-bit data reads.

The data space address is mapped into program memory as shown Figure 1-8. Note that, by incrementing the PC by 2 for each program memory word, the LS 14 bits (15 bits for the TBLRD, TBLWT instructions)

of data and program space addresses directly translate. The remaining bits are provided by the Data Space Program PAGE register, DSPPAG<7:0> as shown in Figure 1-8.

### 1.2.5.1 Data Pre-Fetch from Program Space within a REPEAT loop

When *prefetching* data resident in program space via the data space window from within a REPEAT loop, all iterations of the repeated instruction will reload the instruction from the Instruction Latch without re-fetching it, thereby releasing the program bus for a data prefetch as shown in Figure 1-7. In this example, the initial 2 data words for the first iteration of the instruction to be repeated (MACA) are fetched by a CLRACA instruction. As one of the words resides in program space, an additional cycle is required. The initial fetch of the MACA instruction is performed by the REPEAT instruction.

It is important to note that only the MAC class of instructions, which operate with prefetched data, will operate in this manner. All other instructions (e.g. MOV) which require data to be read by the end of Q2 will require the additional cycle in order to complete the data read prior to execution of the instruction during the second cycle.

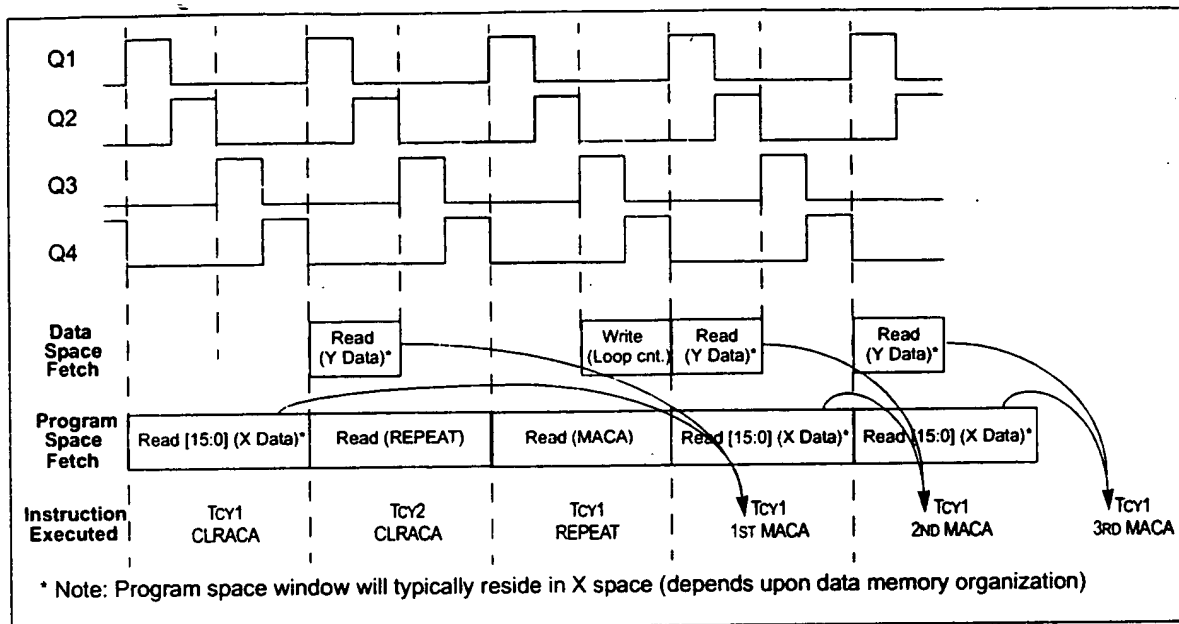


FIGURE 1-7: PS DATA READ THROUGH DS WITHIN A REPEAT LOOP EXAMPLE

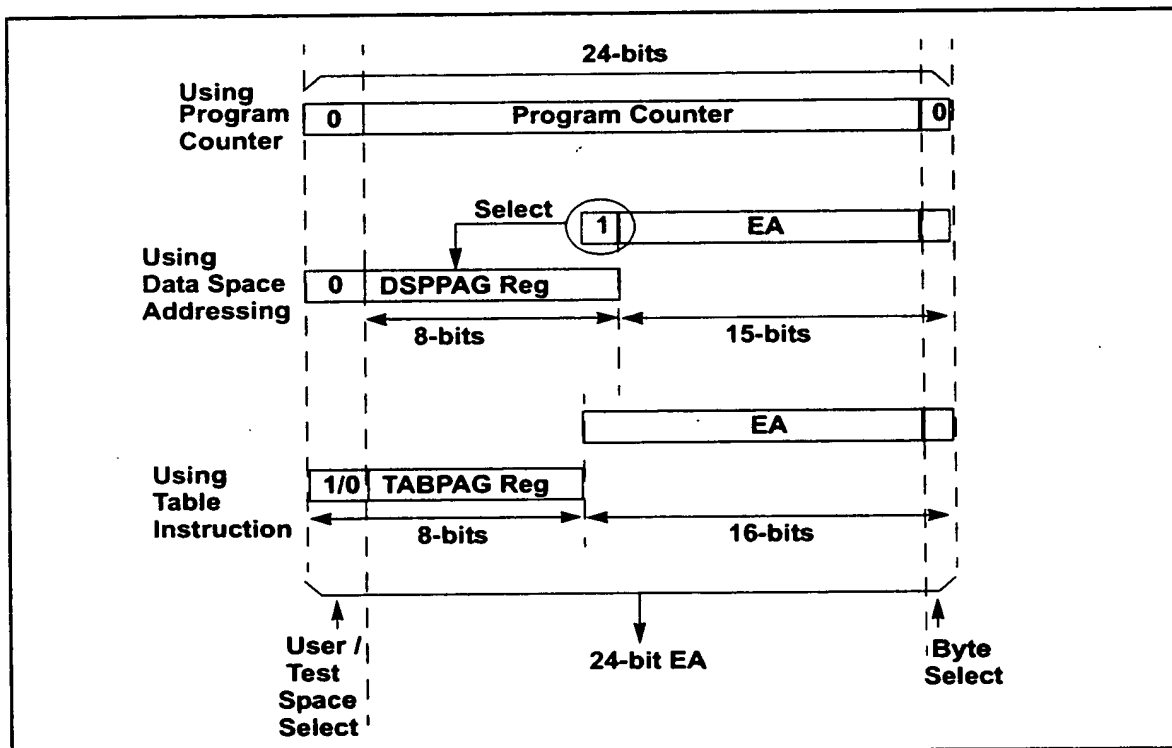


FIGURE 1-8: DATA ACCESS FROM PROGRAM SPACE ADDRESS GENERATION



### 1.3 Program Address Space

The program address space is 8M long words. It is addressable by a 24-bit value from either the PC, table instruction EA or data space EA when program space is mapped into data space as defined by Table 1-1. Note that the program space address is incremented by two between successive program words in order to provide compatibility with data space addressing. Consequently, the LS-bit of the program space address is always 0, resulting in 23-bits (8M) of address. Program space data accesses use the LS-bit of the program space address as a byte select (same as data space).

**Note:** Memory mapped or stacked PC must include the zero LS-bit.

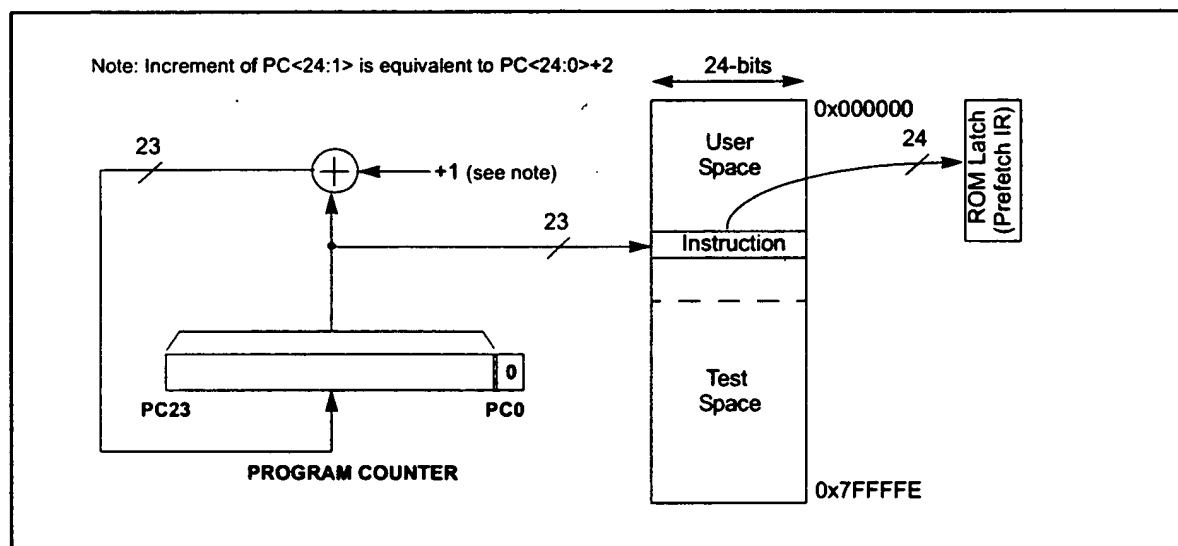
The address space is split into two 4M long word spaces, one for user space the other for test and vector memory space as shown in Figure 1-10. When in user mode, program space access is restricted to the lower 4M long word space, 0x000000 to 0x7FFFFE for all accesses other than TBLRD/TBLWT which use TABPAG[7] to determine user or test space access. Exception vectors also reside in test space. While in user mode, the PC is inhibited from 'rolling over' into test space (i.e. PC[23] is always clear).

Access Type	Access Space	Program Space Address				
		[23]	[22:16]	[15]	[14:1]	[0]
Instruction Access	User	0	PC[23:1]			0
Instruction Access	Test	1	PC[23:1]			0
TBLRD/TBLWT	User/Test	TABPAG[7:0]		Data EA [15:0]		
DS Window into PS	User	0	DSPPAG[7:0]		Data EA [14:0]	
DS Window into PS	Test	Not allowed				

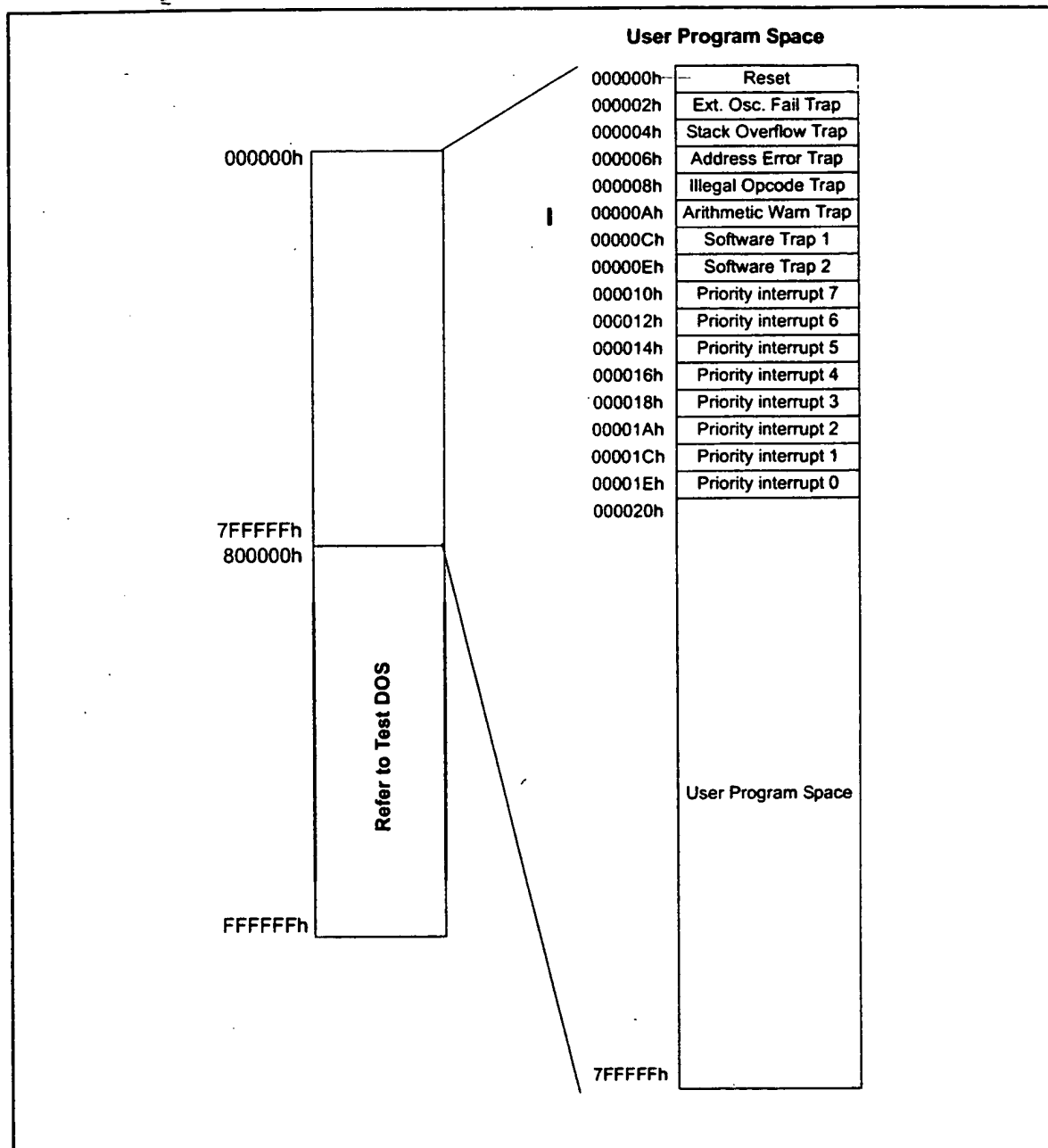
**TABLE 1-1: PROGRAM SPACE ADDRESS CONSTRUCTION**

The program memory width is 24-bits (long word). To support data storage and FLASH programming, the array must support both word wide access from bits 0-15 and byte wide access from bits 16-23.

An instruction fetch example is shown in Figure 1-9. Note that incrementing PC[23:1] by one is equivalent to adding 2 to PC[23:0].



**FIGURE 1-9: INSTRUCTION FETCH EXAMPLE**



**FIGURE 1-10: PROGRAM SPACE MEMORY MAP**

### 1.3.1 Program Space Alignment and Data Access using Table Instructions

This architecture (internally) fetches 24-bit wide program memory. Consequently, instructions are always aligned. However, as the architecture is modified Harvard, data can also be present in program space.

There are two methods by which program space can be accessed - via special TABLE instructions or through the remapping of a 16Kword program space page into the upper half of data space (see Section 1.2.5). The TBLRDL and TBLWTL instructions offers a direct method of reading or writing the LS word of any address within program space without

going through data space which is preferable for some applications. The TBLRDH and TBLWTH instructions are the only method whereby the upper 8-bits of a program word can be accessed as data.

Figure 1-8 shows how the EA is created for table operations.

### 1.3.1.1 Table instructions

A set of TABLE instructions are provided to move byte or word sized data to and from program space. The instructions are orthogonal even though the MS byte will always read zeros. See dsPIC Instruction Set DOS for more details.

1. **TBLRDH:** Table read high  
*Word:* Read the LS word of the program address  
P[15:0] maps to D[15:0]  
*Byte:* Read one of the LS bytes of the program address  
P[7:0] maps to D[7:0] when byte select=0;  
P[15:8] maps to D[7:0] when byte select=1
2. **TBLWRL:** Table write low  
See Program Memory DOS-00204
3. **TBLRDH:** Table read high  
*Word:* Read the MS word of the program address  
P[23:16] maps to D[7:0]; D[15:8] always = 0  
*Byte:* Read one of the MS bytes of the program

address

P[23:16] maps to D[7:0] when byte select=0;

D[7:0] will always = 0 when byte select=1

#### 4. TBLWRH: Table write high

See Program Memory DOS-00204

Where:

P = program space long word

D = data space word

Program space writes (for FLASH programming) have to be performed in a specific order as described in the Program Memory DOS-00204.

The PC is incremented by two for each successive 24-bit program word. This allows program memory addresses to directly map to data space addresses as shown in Figure 1-11. Program memory can thus be regarded as two 16-bit word wide address spaces residing side by side, each with the same address range. TBLRDH and TBLWTH access the space which generates the LS data word, and TBLRDH and TBLWTH access the space which generates the MS data byte. As program memory is only 24-bits wide, the upper byte from this latter space does not exist, though it is addressable. It is therefore termed the 'phantom' byte.

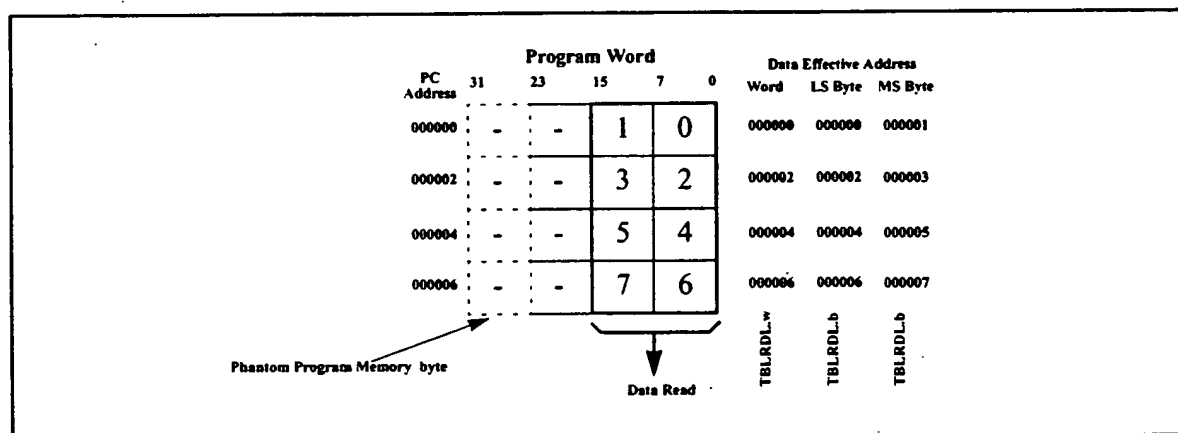


FIGURE 1-11: PROGRAM DATA TABLE ACCESS (LS WORD)

For all the table instructions, the calculated EA (using MODE2 addressing modes) is concatenated with the 8-bit data table page register, TABPAG<7:0>, to form a 23-bit effective programs space address plus a byte select for program memory as shown in Figure 1-8. As there are 15-bits of program space address from the calculated EA, the data table page size in program memory is therefore 32K words.

The LS-bit of the calculated EA becomes the byte select and is used by TBLRDH and TBLWRL (see Program Memory DOS-00204) to select which byte is

accessed. The TBLRDH and TBLWRL instructions therefore view program space as byte or aligned word addressable, 16-bit wide, 64K byte pages (i.e. same as data space). EA[0] is ignored for word wide accesses.

The TBLRDH and TBLWRH instructions are used to access the high order byte of the program address. These instructions also support word or byte access for orthogonality but the high order byte of the program address can only be read from the LS byte as shown in Figure 1-12. The MS-byte of a TBLRDH word read

[illegible]

### 1.3.2 HEX Data File Compatibility

The program space data access described above can be made compatible with HEX format data files by regarding the program memory as 32-bits wide. Inserting the 'phantom' byte as shown in Figure 1-13 allows the HEX format byte address to be directly used as the TBLWTL.w and TBLWTH.(b) EA after a single bit right shift.



can be supported as described in Section 1.2.5.

As discussed in Section 1.3.3, program space is 24-bits wide which will require either a mix of external FLASH devices to provide all 24-bits in one bus cycle, or several cycles to fetch the 24-bit word in either 8-bit

---

or 16-bit sections. The External Bus Interface (EBI) module will attempt to provide the user maximum flexibility in this area.

Data access is potentially somewhat simpler as the fundamental data size is 16-bits. To permit single (bus) cycle, 16-bit wide external memory access, the EBI may optionally be configured to read from a 16-bit external bus and then automatically concatenate an 8-bit trap field prior to passing the 24-bit pword to the CPU. A 16-bit external data bus can therefore be provided for data storage without compromising device robustness. The unused portion of the external bus data path can also revert back to I/O.

## 1.4 Clocking Scheme

Each instruction cycle (Tcy) is comprised of four Q cycles (Q1-Q4). These Q clock are derived using simple logic (i.e. there is no requirement to make them non-overlapping) within the core (and each peripheral module) from global QA and QB quadrature clocks. The quadrature clocks are generated by the PLL module. Maintaining minimal skew between QA and QB across the device will be a critical factor in attaining the target performance. The four phase Q cycles provide the timing/designation for the Decode, Read, Process Data, Write etc., of each instruction cycle. Figure 1-14 shows the relationship of the Q cycles to the instruction cycle for both MCU and DSP instructions. The four Q cycles that make up an execution instruction cycle (Tcy) can be generalized as:

Q1: Instruction Decode Cycle or forced NOP and source EA calculation

Q2: Source Data Read Cycle or NOP

Q3: Process the Data and destination EA calculation

Q4: Destination Data Write Cycle or NOP

Each instruction will show the detailed Q cycle operation for the instruction.

**Note:** Although most instructions follow the scheme above, some issue two reads, others two writes per cycle. See dsPIC Instruction Set DOS for details.

From a Q cycle perspective, the DSP instructions differ from in MCU instruction in so much as the DSP instruction can perform two simultaneous source data reads during the Q1/Q2 access from X and Y data space.

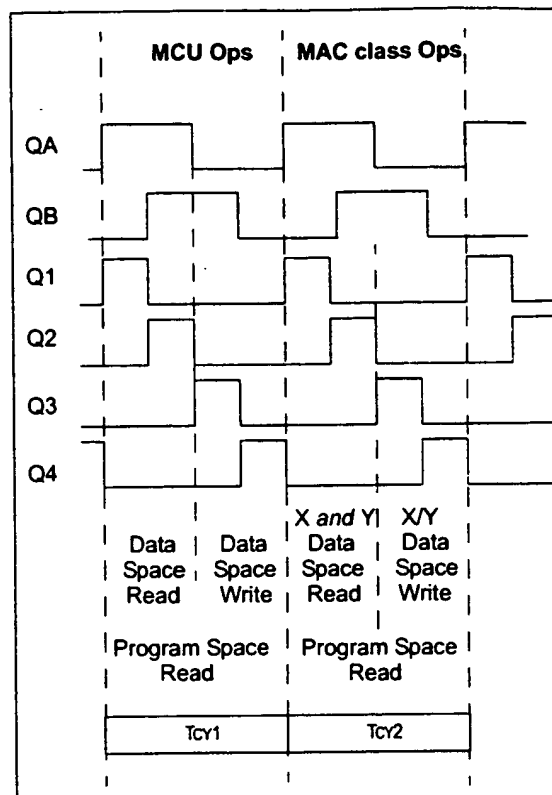


FIGURE 1-14: BASIC CORE TIMING

### 1.4.1 Instruction Cycle Timing

Internally, the program address latch is updated at the start of every Q1, and the instruction is fetched from the program memory and latched into the ROMLATCH using Q4. The PC is actually adjusted (incremented or loaded) during Q4 of the previous cycle but not transferred into the program address latch until the next instruction has started.

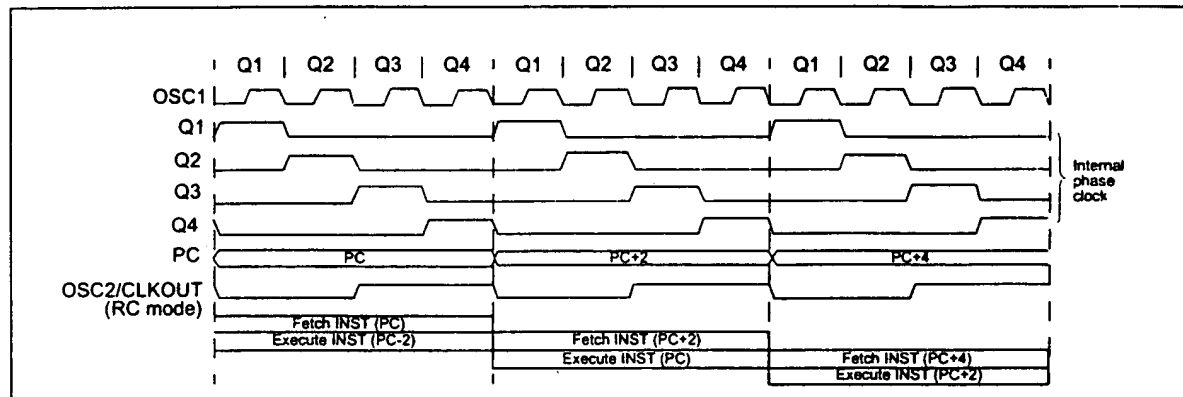
The instruction is decoded and executed during the following Q1 through Q4. The Instruction is decoded during Q1, though some pre-decode of register and addressing mode bit fields during the prior Q4 may be necessary to speed up execution.

**Note:** Care must be taken with any pre-decoding of the instruction to avoid issues (e.g. having to add extra cycles) during interrupt or call returns.

There are two, independent data space accesses to (possibly) two different addresses during each instruction cycle. During Q1 the (remainder) of the instruction decode is performed and the source operand EA is calculated. During Q2, the source operand data is fetched from memory or peripherals. The ALU performs the computation during Q3 at the same time as

The data space buses are addressed twice during each cycle with a read (two reads for the DSP instructions) followed by a write.

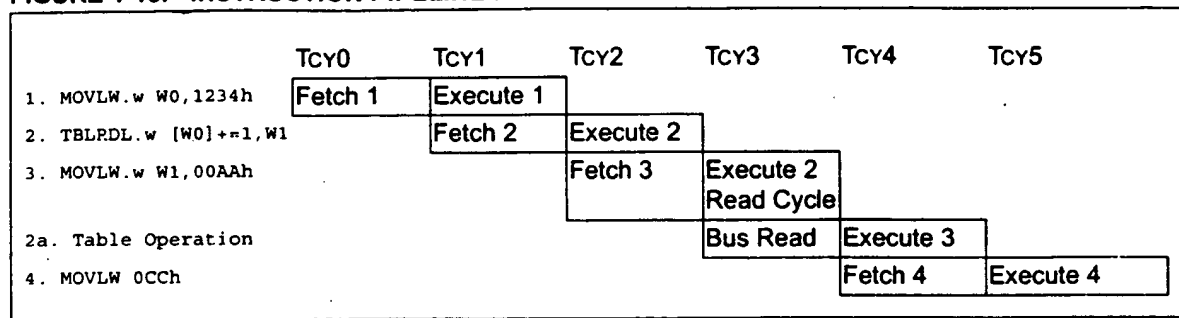
### FIGURE 1-15: CLOCK/INSTRUCTION CYCLE



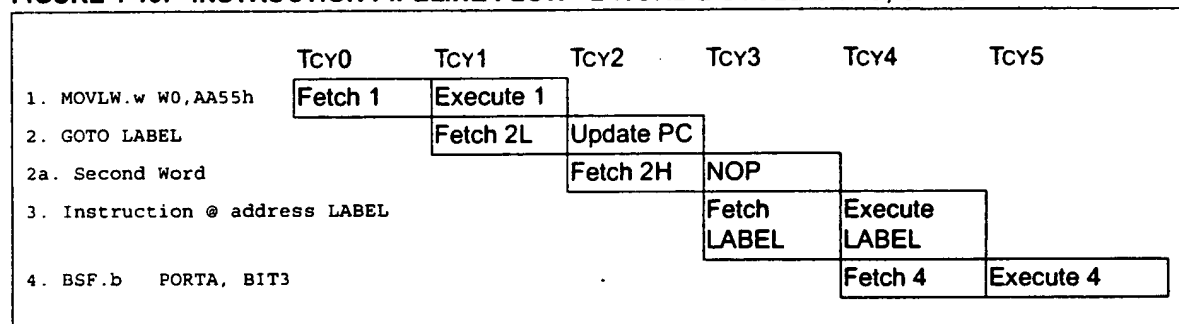




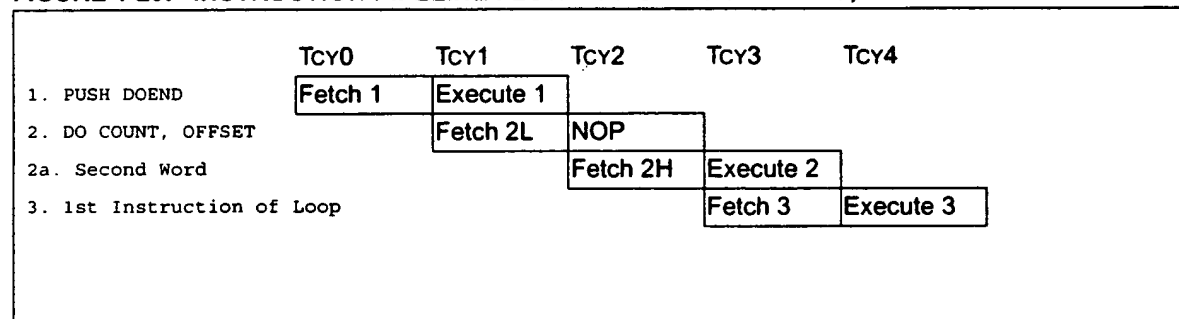
**FIGURE 1-18: INSTRUCTION PIPELINE FLOW - 1 WORD 2 CYCLE TABLE OPERATIONS**



**FIGURE 1-19: INSTRUCTION PIPELINE FLOW - 2 WORD 2 CYCLE GOTO, CALL**



**FIGURE 1-20: INSTRUCTION PIPELINE FLOW - 2 WORD 2 CYCLE DO, DOW**



## 1.5.2 Program Flow Loop Control

The dsPIC core supports both REPEAT and DO instruction constructs to provide unconditional automatic program loop control.

### 1.5.2.1 REPEAT Loop Construct

The REPEAT instruction will cause the instruction immediately following to be repeated a fixed number of times as defined by an 14-bit literal encoded in the instruction. The REPEATW instruction will cause the instruction immediately following it to be repeated a fixed number of times as defined by the contents of a W register declared within the instruction, enabling the loop count to be a variable. The loop count is held in

the 16-bit RCOUNT register (which is memory mapped) and is thus user accessible. It is initialized by the REPEAT[W] instruction during Q2.

The instruction to be repeated is prefetched during the REPEAT[W] instruction and held in the ROMLATCH. It is not fetched again for all subsequent iterations, and the Instruction Register is loaded from the locked ROMLATCH.

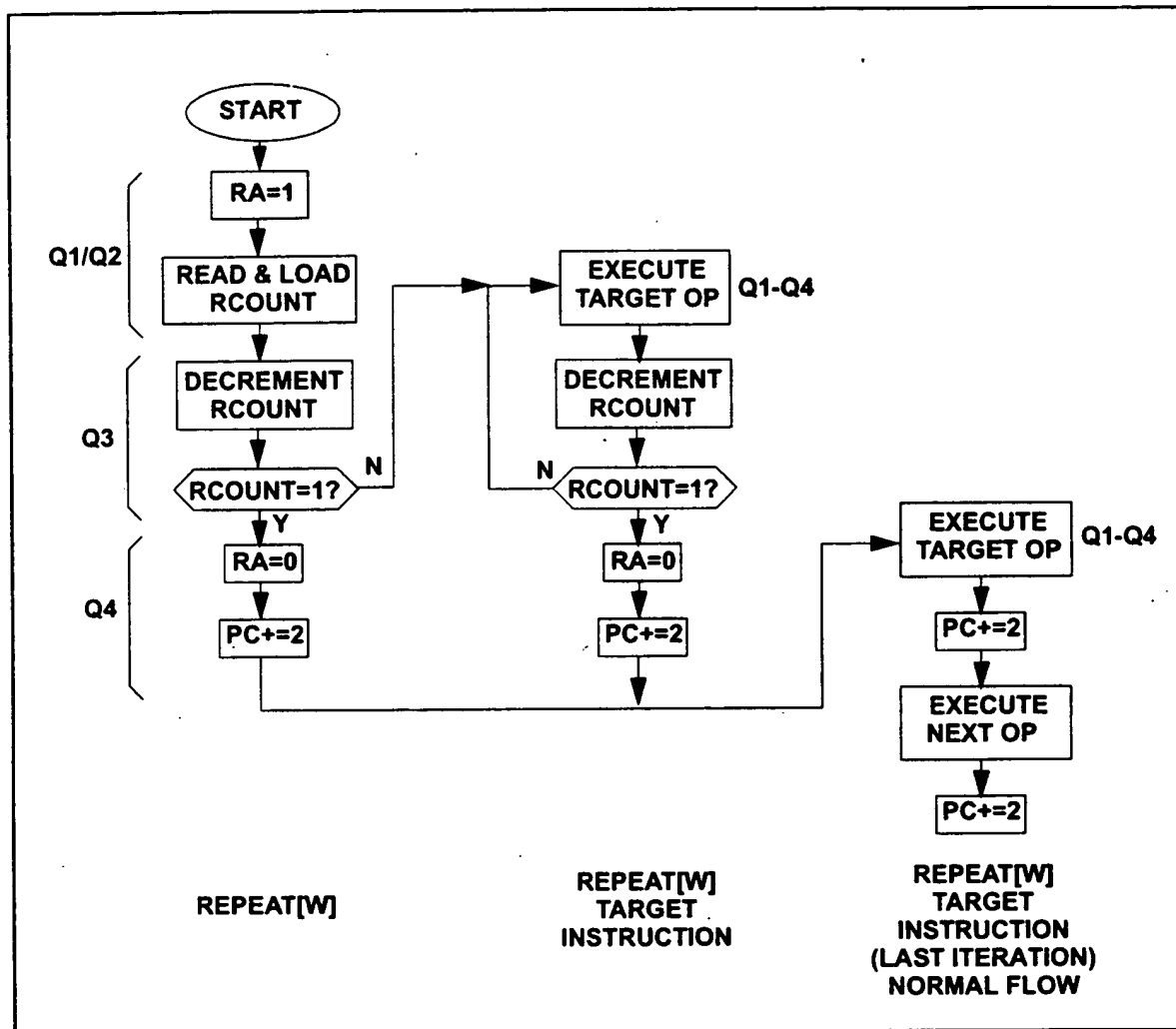
For a loop count value equal 1, REPEAT[W] has the effect of a NOP (other than RCOUNT being loaded with 1). The RA (Repeat Active) status bit in the SR is not set during execution of REPEAT[W] and the PC is incremented as would normally be the case during Q4 of an instruction. The repeat loop is essentially dis-

abled before it begins, allowing the next instruction to execute only once while pre-fetching the subsequent instruction (i.e. normal execution flow).

For loop count values greater than 1, the PC is *not* incremented as would normally be the case during Q4 of an instruction (and will therefore continue to point to the instruction to be repeated). Further PC increments are inhibited until the loop ends. The RA (Repeat Active) status bit in the SR is also set during execution

of REPEAT[W]. See Figure 1-21 for a functional flow diagram of the REPEAT[W] operation, and Figure 1-22 for an instruction pipeline example of a REPEAT[W] loop.

**Note:** RA is a read only bit within the SR and cannot be modified through software.



**FIGURE 1-21: REPEAT[W] LOOP FUNCTIONAL FLOW**

The RCOUNT register is decremented then tested during each instruction iteration. It will equal two at the beginning of the penultimate instruction. The subsequent decrement will make RCOUNT=1, signifying the end of the repeat loop, which causes the RA bit in the SR to be cleared. In addition, the PC increment inhibit is released and the PC bumps in Q4 of this instruction to point to the instruction after the repeated instruction. The last instruction to be repeated is then executed as

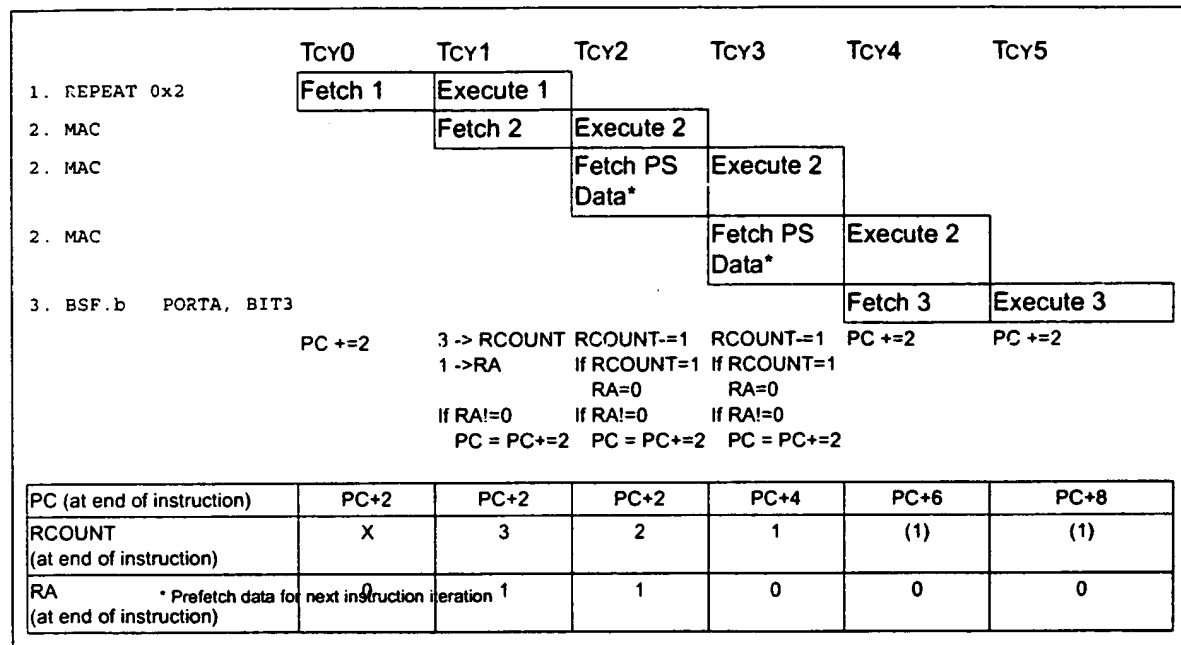
a normal instruction (i.e. includes an instruction prefetch & PC bump). Testing for the end of loop during the penultimate instruction is required to allow a normal instruction prefetch to occur during the last iteration (i.e. no delays due to 'end of loop' tests).

A consequence of executing the last instruction outside the repeat loop is that the loop will effectively iterate [loop count + 1] times (i.e. a loop count of 0 is not

possible). Choosing the loop termination count value to equal one enables the loop count and number of iteration to match for all but RCOUNT equal to zero.

**Note:** For a loop count value of 0, REPEAT will iterate the next instruction 16384 times and REPEATW will iterate the next instruction 65536 times

The combined instruction flow diagram for REPEAT[W] and DO[W] is shown in Figure 1-28..



**FIGURE 1-22: REPEAT[W] INSTRUCTION PIPELINE FLOW**

### 1.5.2.2 REPEAT Loop Interrupt and Nesting

A REPEAT instruction loop may be interrupted at any time. As is the case for all instructions, the PC update is arranged such that it will not be incremented during the instruction when an exception is acknowledged. For a repeated instruction, the PC update is already inhibited (by the RA bit) which ensures that, upon return, the RETFIE instruction will correctly prefetch said instruction (i.e. the stacked PC will point to the instruction to be repeated).

Exception processing proceeds as normal, except for a fast interrupt acknowledgment where the contents of the Instruction Latch are transferred into a temp register (IR Temp). This occurs irrespective of the state of the RA bit and is not related to the REPEAT operation. Standard exception processing completes and the ISR is executed as normal in either case.

Note that, in order to interrupt a REPEAT in progress, the LS-byte of the SR (SRL, which includes the RA bit) is stacked during exception processing. This preserves the state of the RA bit prior to interruption. The RA bit in the SR is then cleared, also during exception processing. In addition, the RCOUNT register has a

shadow register associated with it which is loaded during exception processing (any exception, not just for a fast interrupt). This, in conjunction with the preservation of the RA bit (SRL stacked), permits another REPEAT instruction to be executed within the initial interrupt service routine (i.e. any ISR provided interrupt nesting is not enabled).

Should interrupt nesting be enabled, subsequent interrupts must stack the RCOUNT register before another REPEAT loop may be executed from within the ISR. If RCOUNT is stacked, the RA preservation feature will also operate for all subsequent nested interrupts. Note that RCOUNT must be restored prior to returning from the ISR. The RA bit is restored automatically during interrupt return processing. Also note that for nested interrupts, the most efficient method to handling REPEAT instructions within ISRs will be to always stack RCOUNT.

Interrupt return operates as normal and requires no special handling for returning into a REPEAT[W] loop. Normal interrupts will prefetch the repeated instruction during the second cycle of the RETFIE. Return from a fast interrupt will reload the Instruction Latch from the IR Temp register and execute the next repeat iteration during the second cycle (see Section 5.3.3). The

stacked RA-bit will be restored when the SRL register is popped and, if set, the interrupted REPEAT loop will be resumed.

**Note:** Clearing the RA bit in the stacked SR from within an ISR is a method to force an interrupted loop to terminate (subject to one more iteration) after the interrupt returns. RA is not software modifiable within the SR.

### 1.5.2.3 DO Loop Construct

The DO & DOW instructions will execute instructions following the DO[W] until an end address is reached at which time instruction execution will start again at the instruction immediately following the DO[W]. This will be repeated a finite number of times as defined by either an 14-bit literal encoded in the 1st word of the instruction (for DO) or by the contents of a W register declared within the instruction (for DOW), enabling the loop count to be a variable. The instruction execution order need not be sequential, nor does the loop end address have to be greater than the start address.

Referring to Figure 1-24, the DO[W] instruction loads the loop count value into the loop count register (DCOUNT) during Q2. Note that, as it is required that a REPEAT[W] instruction be executable from within a DO loop, the DCOUNT and RCOUNT registers must

be independent. The associated decremter can be shared however, the last instruction of a DO[W] loop cannot be:

1. a REPEAT[W] instruction or
2. the instruction within a repeat loop.

See Section 1.5.2.6 for details.

**Note:** Ideally, these circumstances should be detected & flagged by the assembler.

The loop start address (PC) is stored in the DOSTART register during Q2 of the second cycle. The two cycle DO[W] instruction then calculates the end address by executing a 23-bit signed addition of the current PC[23:1] (which points to the first loop instruction) and a signed 16-bit literal offset encoded within the 2nd word of the DO[W] instruction. This is executed using the MCU ALU during Q1 and Q3 of the 2nd cycle. The loop end address is stored in the DOEND register during Q4. The DOEND and DOSTART registers are closely coupled with the PC as shown in Figure 1-23. The DA bit within the SR is also set during DO, forcing all subsequent instruction cycles to execute a PC address compare during Q1. This comparison must occur for every cycle (i.e. not just once for a 2 cycle instruction).

**Note:** DO is not required to execute from test memory space. The DOSTART, DOEND registers are therefore restricted to 22-bits each with an additional MS bit always = 0.

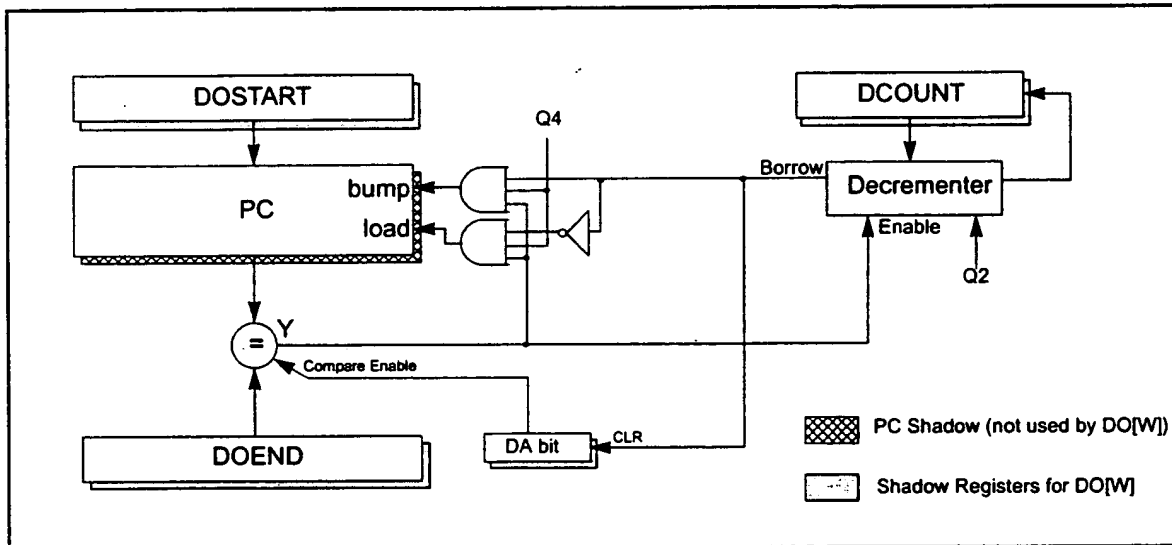
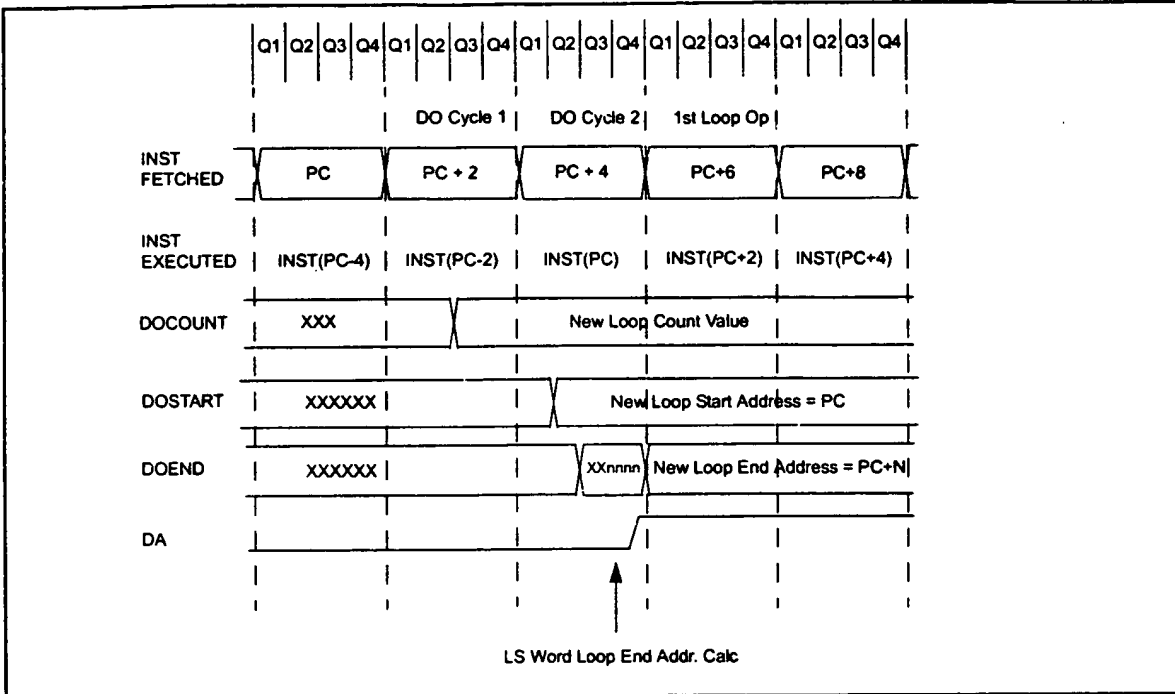


FIGURE 1-23: TOP LEVEL BLOCK DIAGRAM OF DO LOOP HARDWARE OPERATION



**FIGURE 1-24: DO LOOP ENTRY TIMING**

The DO[W] literal address offset is such that the end address is calculated to be the last instruction within the loop. This will cause a valid PC address compare during the Q1 compare operation of the penultimate instruction (i.e. during the prefetch of the last instruction). This will then enable the loop counter to be decremented and tested, and the result combined with the address compare during Q3 of the same instruction.

If the loop counter after decrement does not equal 1 (as shown in Figure 1-25), the PC is loaded with the loop start address during Q4 (such that the last instruction will prefetch the first loop instruction, initiating another loop pass). The loop penultimate instruction does not have to be the one immediately preceding the last loop instruction. It can be a branch or GOTO instruction which targets the last instruction as shown in Figure 1-26 (for a branch).

If the loop counter after decrement equals 1 (as shown in Figure 1-27), then the DA bit in the SR is cleared and the PC is incremented as normal during Q4 (such that the last instruction will prefetch the instruction following it and exit the loop).

The DO loop is equivalent to the 'C' construct DO-WHILE which implies that the loop will be executed at least once. Choosing the loop termination count value to equal one enables the loop count and number of iteration to match for all DCOUNT values except zero.

For a DCOUNT loop count value of 0, DO will iterate the loop 16384 times and REPEATW will iterate the loop 65536 times

**Note:** The loop end comparison is an equality test only. The loop end address must be prefetched in order for the end of loop condition to be recognized. That is, exiting the loop to a PC value greater than the end address (or less than the start address) will not cause the loop count to change.

The combined instruction flow diagram for REPEAT[W] and DO[W] is shown in Figure 1-28.

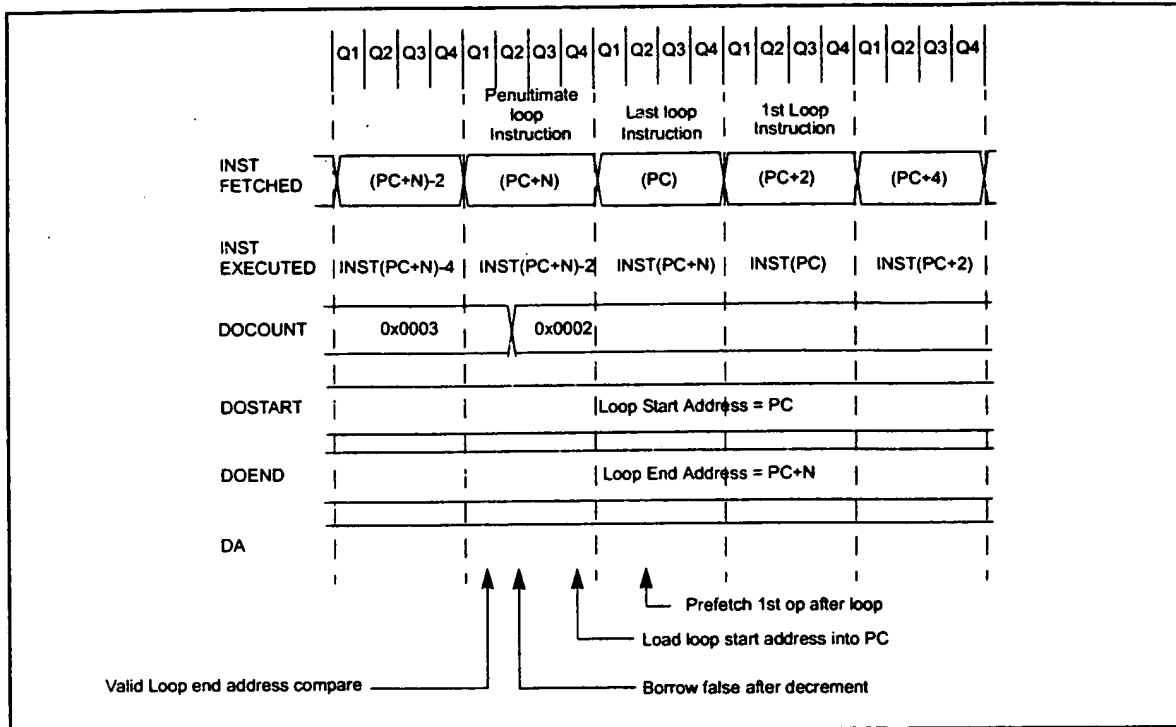


FIGURE 1-25: DO LOOP CONTINUATION TIMING

#### 1.5.2.4 DO Loop Nesting

The DOSTART, DOEND and DCOUNT loop registers have a shadow register associated with them which permit a single level of nesting. In addition, as the DOSTART, DOEND and DCOUNT registers are user accessible, they may be manually saved to permit additional nesting. However, it should be noted that the overhead associated with manually saving these registers outweighs the benefits of additional DO loop nesting with the possible exception of a DO loop within an interrupt (see Section 1.5.2.5).

When a DO is executed, the DOSTART, DOEND and DCOUNT registers are transferred into the shadow registers prior to being updated with the new loop values. The DA bit is also shadowed prior to being set during DO execution. These operations occur for all DO instruction executions, whether nested or not. Similarly, during all loop exits, the shadow contents of the DOSTART, DOEND and DCOUNT registers and the DA bit are transferred back into their respective host registers.

#### 1.5.2.5 DO Loops and Interrupts

A DO[W] loop may be interrupted at any time without penalty.

Note that, in order to suspend an interrupted DO loop during execution of an ISR, the LS-byte of the SR (SRL, which includes the DA bit) is stacked then cleared (in the SR) during exception processing. Although this is not essential because the DO loop end address is unlikely to be encountered during the ISR, it is consistent with REPEAT operation. If a background DO loop was active (stacked DA bit set), the DOSTART, DOEND and DCOUNT registers must then be stacked before another DO loop may be executed from within the ISR. This applies to any interrupt class. These registers must be restored prior to returning from the ISR.

**Note:** Prior to executing a DO within an interrupt requires stacking and restoring 5 words of data. This overhead may mean DO is not the most efficient means for loop control within an ISR.

Interrupt return operates as normal and requires no special handling for returning into a DO[W] loop. The stacked DA bit will be restored into the SRL register and, if set, the interrupted DO loop will resume.

**Note:** Clearing the DA bit in the stacked SR from within an ISR is a method to force an interrupted loop to terminate early after the interrupt returns. The loop will complete the iteration underway and then terminate. If the

interrupt occurs during the penultimate or last instruction of the loop, one more iteration of the loop will occur. DA is not software modifiable within the SR.

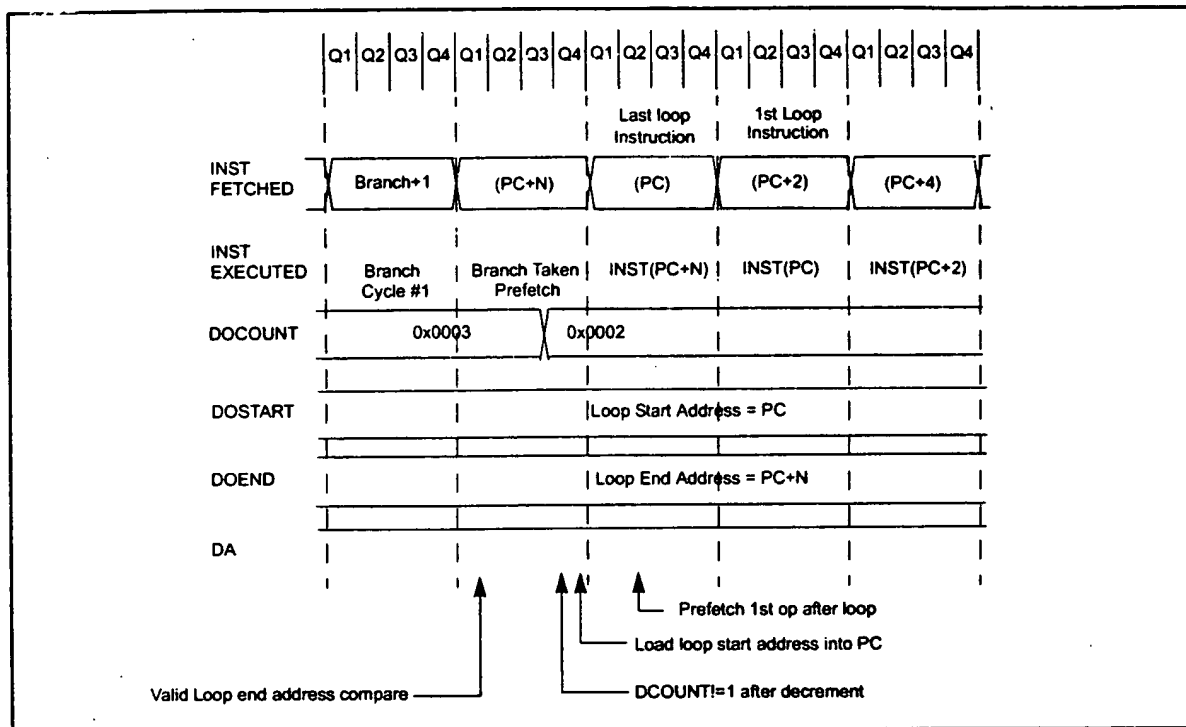


FIGURE 1-26: DO CONTINUE TIMING WITH BRANCH TO LAST INSTRUCTION

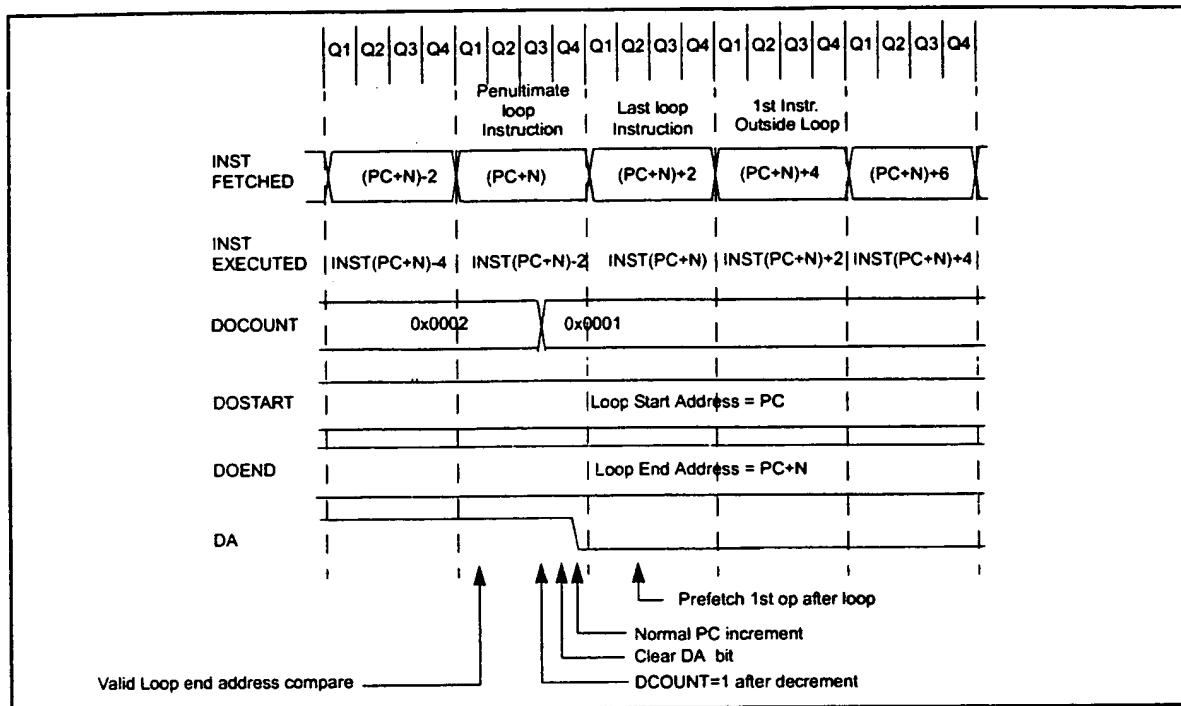


FIGURE 1-27: DO LOOP EXIT TIMING



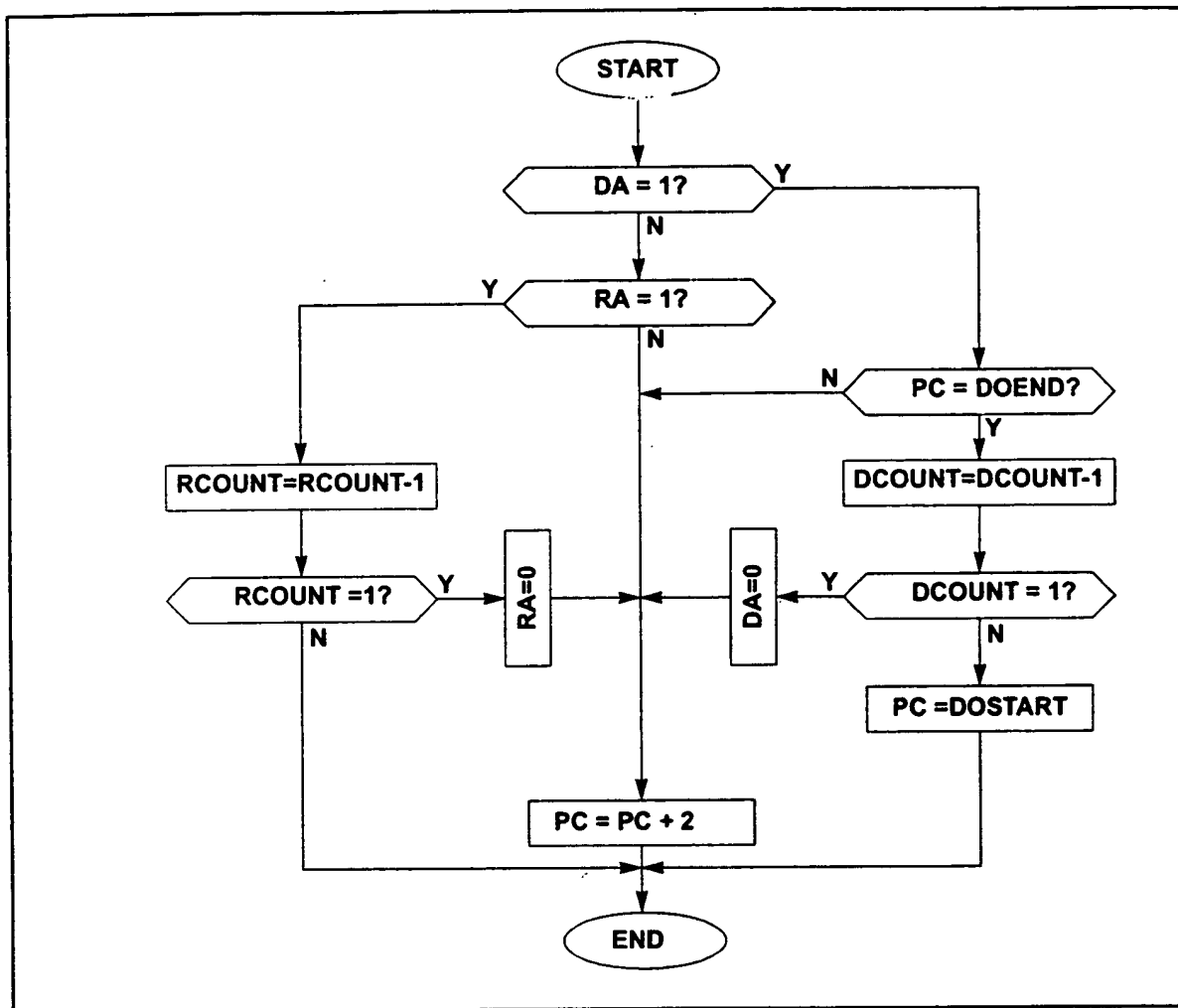


FIGURE 1-28: DO AND REPEAT FLOW DIAGRAM

#### 1.5.2.6 DO and REPEAT Restrictions

Any instruction can follow a REPEAT except for:

1. Flow control (any branch, compare and skip, GOTO, CALL, CALLW, RCALL, RETURN or RETLW)
2. Another REPEAT or DO

As it is not especially useful to execute any of these instructions within a repeat loop, the restrictions on this instruction are minimal.

REPEAT is interruptible and can be then be nested from within an initial (first, unnested) ISR. If interrupt nesting is enabled, REPEAT can be nested from within any ISR but only after the user stacks the appropriate registers manually (all REPEAT control registers are user accessible).

All DO loops must contain at least 2 instructions because the loop termination tests are performed in the penultimate instruction. REPEAT should be used for single instruction loops. All other restrictions with regard to the DO loop revolve around the last instruction. With the notable exception of CALLW, the last instruction should not be:

1. Flow control (any branch, compare and skip, GOTO, RCALL)
2. Another REPEAT or DO
3. Instruction within a repeat loop
4. Any 2 word instruction

If at all possible, the assembler should be capable of flagging these instructions if placed at the end of a DO loop.

The (one word) `CALLW` will function correctly at the end of a `DO` loop because the stacked PC will address the start of loop instruction (to fetch upon return).

PC relative instructions (e.g. `RCALL`, branches) won't work correctly at the end of a loop because the PC calculation will be performed using the current PC value which will be the loop start address. That is, the assembler psuedo-PC and the real PC do not match at this point.

Should execution of a `REPEAT[W]` instruction as the last loop instruction be attempted, the `DO[W]` loop counter will take priority and the `REPEAT` target instruction will never be executed before the `DO[W]` loop jumps to the loop start. Should the last loop instruction be the instruction being repeated within a `REPEAT` loop, the `DO[W]` loop counter will also take priority and the `REPEAT` target instruction will only execute once with no change to `RCOUNT` before the `DO[W]` loop jumps to the loop start.

Two word instructions will fail if placed at the end of a `DO` loop because the PC is adjusted in the penultimate instruction in order to accommodate the instruction prefetch (without a dead cycle). Consequently, the second word of a two word instruction would therefore be incorrectly fetched from the loop start address.

`RETURN` and `RETLW` will work correctly when the last instruction of a `DO` loop but the user must be responsible for returning into the loop to complete it.

## 1.6 Programmer Model

The programmers model is shown in Figure 1-33 and consists of 16 x 16-bit working registers, 2 x 40-bit accumulators, status register, data table page register, data space program page register, DO and REPEAT registers, and program counter. The working registers can act as data, address or offset registers. All registers are memory mapped (see xxxx).

Most of these registers have a shadow register associated with them as shown in Figure 1-33. The shadow register is used as a temporary holding register and can transfer its contents to or from its host register upon some event occurring. None of the shadow registers are accessible directly. The following rules apply to register transfer into and out of shadows.

- Fast Interrupts entry & exit
  - W0 to W14 shadows transferred
  - PC shadow transferred
  - TABPAG & DSPPAG shadows transferred
  - RCOUNT shadow transferred
  - SR[6:0] shadow bits transferred
- Normal Interrupt Entry
  - RCOUNT shadow transferred
  - SR[6] shadow bit transferred
- Nested DO
  - DOSTART, DOEND, DCOUNT shadows loaded

Byte instructions which target the working register array only effect the least significant byte of the target register. However, a consequence of memory mapped working registers is that both the least *and* most significant bytes can be manipulated through byte wide data memory space accesses.

### 1.6.1 Uninitialized W Register Trap

The W register array (except W15) is not effected by a reset and therefore must be considered uninitialized until a written to. An attempt to read an uninitialized register for an address access will generate an address error trap (fetch of an uninitialized address). In this situation, the user will most likely choose to reset the application, though recovery may be possible through an examination of the problematic instruction (via the stacked return address).

This function is achieved through the addition of a single latch to each W register (W0 through W14). The latch is cleared by reset and set by the first write to the associated register, as shown in Figure 1-29. When the latch is clear, a read of the corresponding register to either AGU will force an address error trap. W15 is initialized during reset (see Section 1.6.3) and consequently does not require this feature.

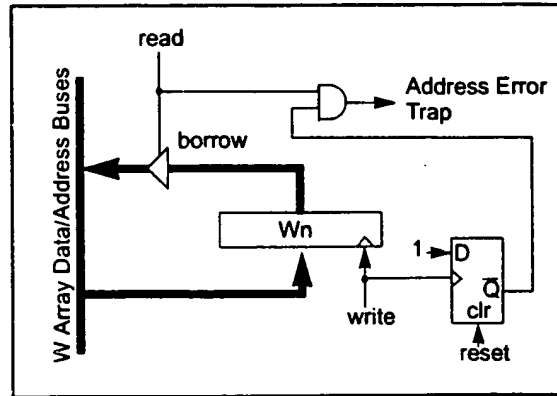


FIGURE 1-29: UNINITIALIZED W REGISTER TRAP

### 1.6.2 Default W Register Selection

The default W register for all file register instructions is defined by the WD[3:0] field in the CORCON (CORE CONTROL register). This field is reset to 0x0000, corresponding to register W0. As most of the CORCON function relates to DSP operation, it is discussed in Section 2.0, DSP Engine.

### 1.6.3 Software Stack Pointer

W15 has been dedicated as the software stack pointer, and will be automatically modified by exception processing and subroutine calls and returns. However, W15 can be referenced by any instruction in the same manner as all other W registers. This simplifies reading, writing and manipulating the stack pointer (e.g. creating stack frames).

**Note:** In order to protect against misaligned stack accesses, W15[0] is always clear.

W15 is initialized to 0x0200 during a reset. This will point to valid RAM in all derivatives and will guarantee stack availability for non-maskable trap exceptions or priority level 7 interrupts which may occur before the SP is set to where the user desires it. The user may reprogram the SP during initialization to any location within data space.

W14 has been dedicated as a stack frame pointer as defined by the LNK and ULNK instructions. However, W14 can be referenced by any instruction in the same manner as all other W registers.

The stack pointer always points to the first available free word and fills working from lower towards higher addresses. It pre-decrements for stack pops (reads) and post increments for stack pushes (writes) as shown in Figure 1-32. Note that for a PC push during any CALL instruction, the MS-byte of the PC is zero

extended before the push, ensuring that the MS-byte is always clear. The stack timing is shown in Figure 1-31.

**Note:** A PC push during exception processing will concatenate the SRL register to the MS-byte of the PC prior to the push.

#### 1.6.4 Stack Pointer Overflow Trap

There is a stack limit register (SPLIM) associated with the stack pointer that is uninitialized at reset. SPLIM[15:1] is a 15-bit register. As is the case for the stack pointer, SPLIM[0] is forced to 0 because all stack operations must be word aligned.

The stack overflow check will not be enabled until a word write to SPLIM occurs after which time it can only be disabled by a reset. All EA's generated using W15

as Wsrc or Wdst (but not Wb) are compared against the value in SPLIM. Should the EA be greater than the contents of SPLIM, then a stack error trap is generated. This comparison is a subtraction, so the trap will occur for any SP greater than SPLIM. In addition, should the SP EA calculation wrap over the end of data space (0xFFFF), AGU X will generate a carry signal which will also cause a stack error trap (if the SPLIM register has been initialized).

#### 1.6.5 Stack Pointer Underflow Trap

The stack is initialized to 0x0200 during reset. A simple stack underflow mechanism is provided which will initiate a stack error trap should the stack pointer address ever be less than 0x0200.

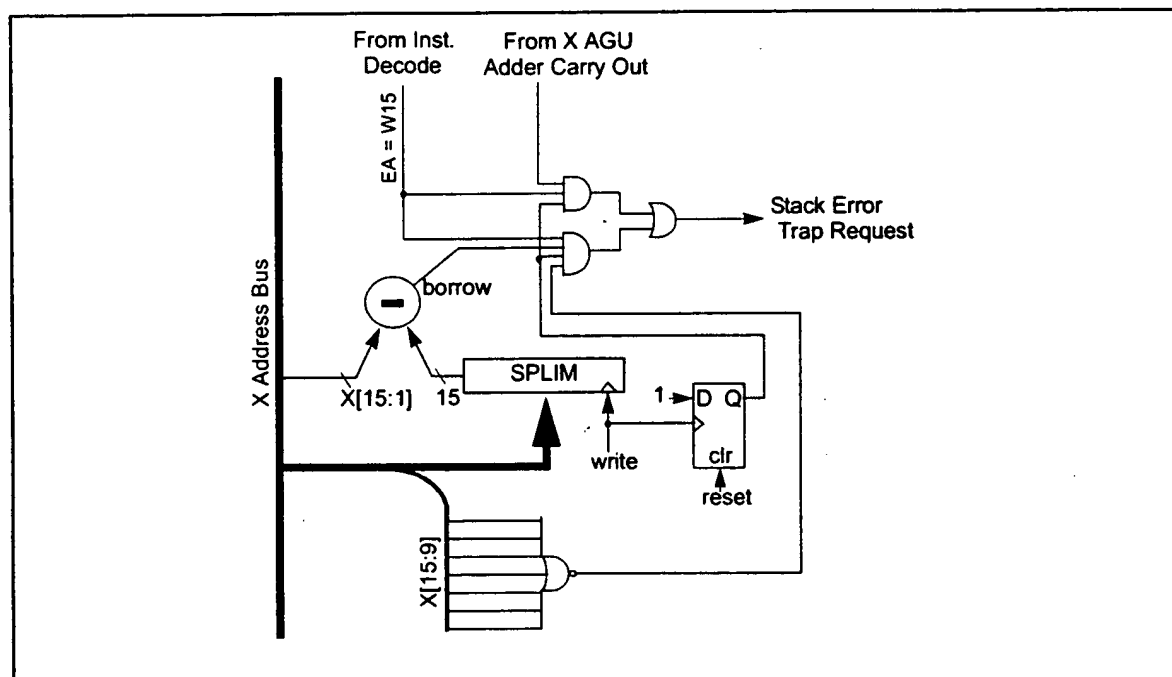


FIGURE 1-30: STACK POINTER OVERFLOW & UNDERFLOW TRAP BLOCK DIAGRAM

#### 1.6.6 Status Register

The dsPIC core has a 16-bit status register (SR), the LS-byte of which is referred to as the lower status register (SRL). A detailed description is shown in Register 1-1.

SRL contains all the MCU ALU operation status flags (including the new 'sticky Z' (SZ) bit) plus the REPEAT and DO loop active status bits. During exception processing, SRL is concatenated with the MS-byte of the PC to form a complete word value which is then stacked.

The upper byte of the SR contains the DSP Adder/Subtractor status bits.

All SR bits are read/write except for the DA and RA bits which are read only because accidentally setting them could cause erroneous operation (include inhibiting PC increments). When the memory mapped SR is the destination address for an operation which affects any of the SR bits, data writes are disabled to all bits.

**REGISTER 1-1: SR, CPU STATUS REGISTER (0XXXXX)****Upper Half:**

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U	U
OA	OB	SA	SB	OAB	SAB	-	-
bit 15						bit 8	

**Lower Half:**

R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
DA	RA	SZ	N	OV	Z	DC	C
bit 7							bit 0

- bit 15 **OA:** Accumulator A Overflow Status  
1 = Accumulator A overflowed  
0 = Accumulator A not overflowed
- bit 14 **OB:** Accumulator B Overflow Status  
1 = Accumulator B overflowed  
0 = Accumulator B not overflowed
- bit 13 **SA:** Accumulator A Saturation 'Sticky' Status  
1 = Accumulator A is saturated or has been saturated at some time  
0 = Accumulator A is not saturated
- bit 12 **SB:** Accumulator B Saturation 'Sticky' Status  
1 = Accumulator B is saturated or has been saturated at some time  
0 = Accumulator B is not saturated
- bit 11 **OAB:** OA || OB Combined Accumulator Overflow Status  
1 = Accumulators A or B have overflowed  
0 = Neither Accumulators A or B have overflowed
- bit 10 **SAB:** SA || SB Combined Accumulator 'Sticky' Status  
1 = Accumulators A or B are saturated or have been saturated at some time in the past  
0 = Neither Accumulator A or B are saturated
- bit 9-8 **Unused**
- bit 7 **DA:** DO Loop Active  
1 = DO loop in progress  
0 = DO loop not in progress
- bit 6 **RA:** REPEAT Loop Active  
1 = REPEAT loop in progress  
0 = REPEAT loop not in progress
- bit 5 **SZ:** MCU ALU 'sticky' Zero bit  
1 = An operation which effects the Z bit has set it at some time in the past  
0 = The most recent operation which effects the Z bit has cleared it (i.e. a non-zero result)
- bit 4 **N:** MCU ALU Negative bit
- bit 3 **OV:** MCU ALU Overflow bit
- bit 2 **Z:** MCU ALU Zero bit
- bit 1 **DC:** MCU ALU Half Carry/Borrow bit
- bit 0 **C:** MCU ALU Carry/Borrow bit

**Legend**

R = Readable bit

W = Writable bit

U = Unimplemented bit, read as '0'

-n = Value at POR

1 = bit is set

0 = bit is cleared

x = bit is unknown

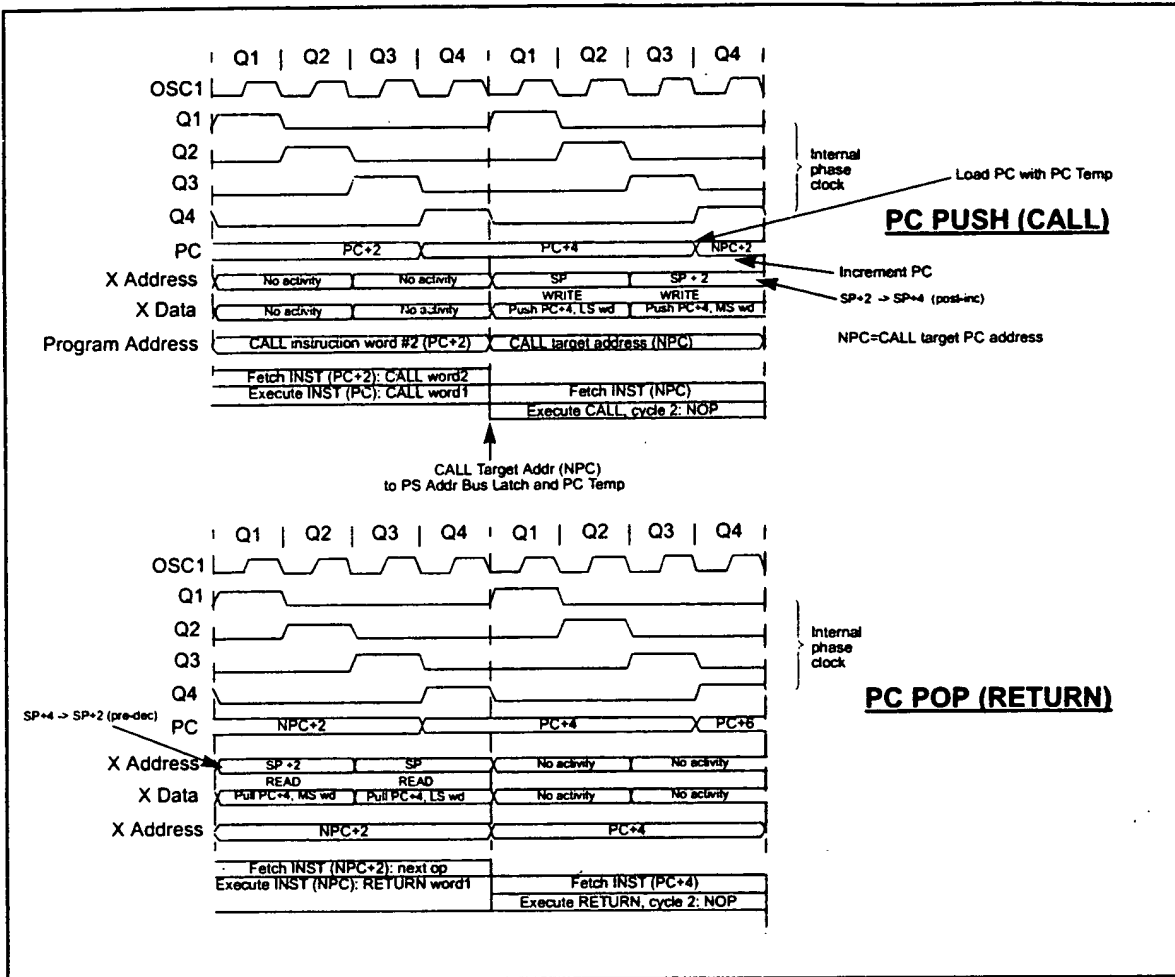


FIGURE 1-31: STACK TIMING EXAMPLE

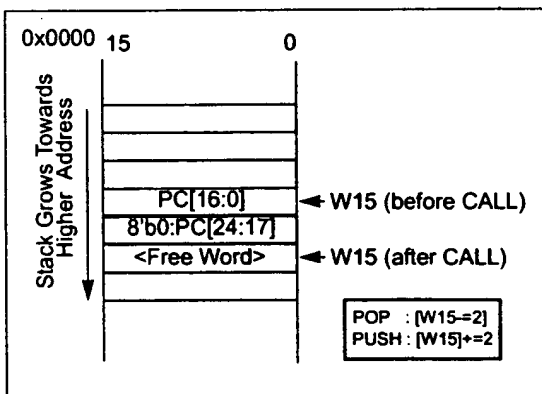


FIGURE 1-32: CALL STACK FRAME

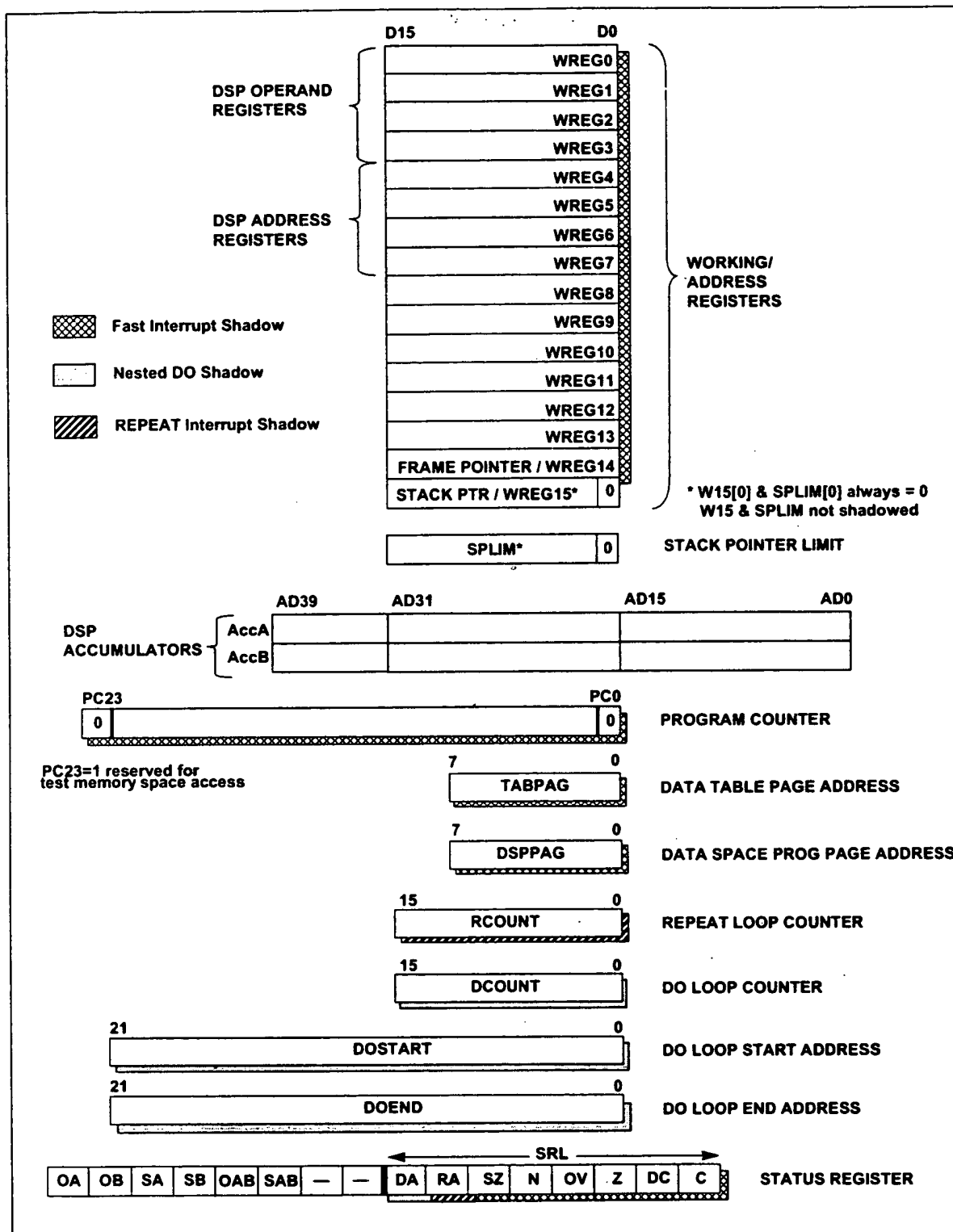


FIGURE 1-33: PROGRAMMERS MODEL

---

## 1.7 Exceptions and Stack

The core supports a prioritized interrupt and trap exception scheme. There are up to 8 levels of interrupt priority, each of which has an interrupt vector associated with it. Each interrupt source is user programmable with regard to what priority (and therefore vector address) it uses. The highest priority interrupt is non-maskable.

There are 7 traps available to improve operational robustness, all of which are non-maskable. They adhere to a predefined priority scheme.

Stacking associated with exceptions and subroutine calls is executed on a software stack. Register W15 is dedicated as the stack pointer and has the LSB = 0.

Refer to Section 5.0 for more complete details of the exception structure.



## TABLES

09870457-060101

TABLE 1-1: DATA BOOK INSTRUCTION SET

Instr #	Assembly Mnemonic	Assembly Syntax	Example	Description	PLA Mnemonic	Page #
1	ADD	ADD A	ADD A	Add Accumulators	ADDAB	62
		ADD f	ADD RAM100	f = f + Ww	ADDWF	69
		ADD f,Ww	ADD RAM100,Ww	Ww = f + Ww	ADDWF	69
		ADD Slit10,Wn	ADD #0xAA,W11	Wd = Slit10 + Wd	ADDLW	68
		ADD Wb,Ws,Wd	ADD W7,[W12++],[W11]--	Wd = Wb + Ws	ADD	61
2	ADDC	ADD Wb,lit5,Wd	ADD W7,#25,[W11]--	Wd = Wb + lit5	ADDLS	67
		ADD A,Wso,Slit4	ADD A,[W12++],#6	16-bit Signed Add to Accumulator	ADDAC	63
		ADDC f	ADDC RAM100	f = f + Ww + (C)	ADDWFC	70
		ADDC f,Ww	ADDC RAM100,Ww	Ww = f + Ww + (C)	ADDWFC	70
		ADDC Slit10,Wn	ADDC #0xAA,W11	Wd = Slit10 + Wd + (C)	ADDCLW	66
3	AND	ADDC Wb,Ws,Wd	ADDC W7,[W12++],[W11]--	Wd = Wb + Ws + (C)	ADDC	64
		ADDC Wb,lit5,Wd	ADDC W7,#25,[W11]--	Wd = Wb + lit5 + (C)	ADDCLS	65
		AND f	AND RAM100	f = f .AND. Ww	ANDWF	74
		AND f,Ww	AND RAM100,Ww	Ww = f .AND. Ww	ANDWF	74
		AND Slit10,Wn	AND #0xAA,W11	Wd = Slit10 .AND. Wd	ANDLW	73
4	ASR	AND Wb,Ws,Wd	AND W7,[W12++],[W11]--	Wd = Wb .AND. Ws	AND	71
		AND Wb,lit5,Wd	AND W7,#25,[W11]--	Wd = Wb .AND. lit5	ANDLS	72
		ASR f	ASR RAM100	f = Arithmetic Right Shift f	ASRF	76
		ASR f,Ww	ASR RAM100,Ww	Ww = Arithmetic Right Shift f	ASRF	76
		ASR Ws,Wd	ASR [W12++],[W11]--	Wd = Arithmetic Right Shift Ws	ASR	75
5	BCLR	ASR Wd,Wrs,Wnd	ASR W0,W2,W1	Wnd = Arithmetic Right Shift Wb by Wrs	ASRW	78
		ASR Wd,lit5,Wnd	ASR W0,#23,W1	Wnd = Arithmetic Right Shift Ws by lit5	ASRK	77
		BCLR.b f,bit3	BCLR.b RAM100,#5	Bit Clear f	BCLRF	81
		BCLR Ws,bit4	BCLR [W12++],#9	Bit Clear Ws	BCLR	80

TABLE 1-1: DATA BOOK INSTRUCTION SET (CONTINUED)

Instr #	Assembly Mnemonic	Assembly Syntax	Example	Description	PLA Mnemonic	Page #
6	BRA	BRA C,Slit16	BRA C,label	Branch if Carry	BC	79
		BRA GE,Slit16	BRA GE,label	Branch if greater than or equal	BGE	82
		BRA GEU,Slit16	BRA GEU,label	Branch if unsigned greater than or equal	BC	79
		BRA GT,Slit16	BRA GT,label	Branch if greater than	BGT	83
		BRA GTU,Slit16	BRA GTU,label	Branch if unsigned greater than	BGTU	84
		BRA LE,Slit16	BRA LE,label	Branch if less than or equal	BLE	85
		BRA LEU,Slit16	BRA LEU,label	Branch if unsigned less than or equal	BLEU	86
		BRA LT,Slit16	BRA LT,label	Branch if less than	BLT	87
		BRA LTU,Slit16	BRA LTU,label	Branch if unsigned less than	BNC	89
		BRA N,Slit16	BRA N,label	Branch if Negative	BN	88
		BRA NC,Slit16	BRA NC,label	Branch if Not Carry	BNC	89
		BRA NN,Slit16	BRA NN,label	Branch if Not Negative	BNN	90
		BRA NOV,Slit16	BRA NOV,label	Branch if Not Overflow	BNV	91
		BRA NZ,Slit16	BRA NZ,label	Branch if Not Zero	BNZ	92
		BRA OA,Slit16	BRA OA,label	Branch if accumulator A overflow	BOA	93
		BRA OB,Slit16	BRA OB,label	Branch if accumulator B overflow	BOB	94
		BRA OV,Slit16	BRA OV,label	Branch if Overflow	BOV	95
		BRA SA,Slit16	BRA SA,label	Branch if accumulator A saturated	BSA	98
		BRA SB,Slit16	BRA SB,label	Branch if accumulator B saturated	BSB	99
		BRA Slit16	BRA label	Branch Unconditionally	BRA	96
		BRA Z,Slit16	BRA Z,label	Branch if Zero	BZ	114
7	BSET	BSET.b f,bit3	BSET.b RAM100,#5	Computed Branch	BRAW	97
		BSET Ws,bit4	BSET [W12++],#9	Bit Set f	BSETF	101
8	BSW	BSW.C Ws,Wb	BSW.C [W12++],W7	Bit Set Ws	BSET	100
		BSW.Z Ws,Wb	BSW.Z [W12++],W7	Write C or Z bit to Ws<Wb>	BSW	102
9	BTG	BTG.b f,bit3	BTG.b RAM100,#5	Write C or Z bit to Ws<Wb>	BSW	102
		BTG Ws,bit4	BTG [W12++],#9	Bit Toggle f	BTGF	106
10	BTSC	BTSC.b f,bit3	BTSC.b RAM100,#5	Bit Toggle Ws	BTG	105
		BTSC Ws,bit4	BTSC [W12++],#9	Bit Test f, Skip if Clear	BTFS	103
11	BTSS	BTSS.b f,bit3	BTSS.b RAM100,#5	Bit Test Ws, Skip if Clear	BTSC	107
		BTSS Ws,bit4	BTSS [W12++],#9	Bit Test f, Skip if Set	BTFS	104
12	BTST	BTST.b f,bit3	BTST.b RAM100,#5	Bit Test Ws, Skip if Set	BTSS	108
		BTST.C Ws,bit4	BTST.C [W12++],#9	Bit Test f	BTSTF	110
		BTST.Z Ws,bit4	BTST.Z [W12++],#9	Bit Test Ws to C or Z	BTST	109
		BTST.C Ws,Wb	BTST.C [W12++],#9	Bit Test Ws to C or Z	BTST	109
		BTST.C Ws,Wb	BTST.C [W12++],W7	Bit Test Ws<Wb> to C or Z	BTSTW	113
		BTST.Z Ws,Wb	BTST.Z [W12++],W7	Bit Test Ws<Wb> to C or Z	BTSTW	113

TABLE 1-1: DATA BOOK INSTRUCTION SET (CONTINUED)

Instr #	Assembly Mnemonic	Assembly Syntax	Example	Description	PLA Mnemonic	Page #
13	BTSTS	BTSTS.b f,bit3 BTSTS.C Ws,bit4 BTSTS.Z Ws,bit4	BTSTS.b RAM100,#5 BTSTS.C [W12++],#9 BTSTS.Z [W12++],#9	Bit Test then Set f Bit Test Ws to C or Z then Set Bit Test Ws to C or Z then Set	BTSTSF BTSTS BTSTS	112 111 111
14	CALL	CALL lii23 CALL.S lii23 CALL Wn CALL.S Wn CLR f CLR Ww CLR Ws	CALL label CALL.S label CALL W11 CALL.S W11 CLR RAM100 CLR Ww CLR [W11]--	Call subroutine Call subroutine Call indirect subroutine Call indirect subroutine f = 0x0000 Ww = 0x0000 Ws = 0x0000	CALL CALL CALLW CALLW CLRF CLRF CLR	115 115 116 116 120 120 117
15	CLR	CLR A,Wxp,Wx,Wyp,Wy,AWB	CLR A,W0,[W5]--4,W1,[W7]--2,W9	Clear Accumulator	CLRAC	118
16	CLRWD	CLRWD	CLRWD	Clear Watchdog Timer	CLRWD	121
17	COM	COM f COM f,Ww COM Ws,Wd	COM RAM100 COM RAM100,Ww COM [W12++],[W11]--	f = f Ww = f Wd = Ws	COMF COMF COM	123 123 122
18	CP	CP f CP Wb,lii5 CP Wb,Ws	CP RAM100 CP W7,#25 CP W7,[W12++]	Compare f with Ww Compare Wb with lii5 Compare Wb with Ws	CPF CPLS CP	129 137 137
19	CP0	CP0 f CP0 Ws	CP0 RAM100 CP0 [W11]--	Compare f with 0x0000 Compare Ws with 0x0000	CPF0 CP0	130 125
20	CP1	CP1 f CP1 Ws	CP1 RAM100 CP1 [W11]--	Compare f with 0xFFFF Compare Ws with 0xFFFF	CPF1 CP1	131 126
21	CPB	CPB f CPB Wb,lii5 CPB Wb,Ws	CPB RAM100 CPB W7,#25 CPB W7,[W12++]	Compare f with Ww Compare Borrow Wb with lii5 Compare Borrow Wb with Ws	CPFB CPBLS CPB	132 128 127
22	CPFSEQ	CPFSEQ f	CPFSEQ RAM100	Compare f with Ww, skip if =	CPFSEQ	133
23	CPFSGT	CPFSGT f	CPFSGT RAM100	Compare f with Ww, skip if >	CPFSGT	134
24	CPFSLT	CPFSLT f	CPFSLT RAM100	Compare f with Ww, skip if <	CPFSLT	135
25	CPFSNE	CPFSNE f	CPFSNE RAM100	Compare f with Ww, skip if ≠	CPFSNE	136
26	DAW.B	DAW.B Wn	DAW.B W11	Wn = decimal adjust Wn	DAW	138
27	DEC	DEC f DEC f,Ww DEC Ws,Wd	DEC RAM100 DEC RAM100,Ww DEC [W12++],[W11]--	f = f - 1 Ww = f - 1 Wd = Ws - 1	DEC DEC DEC	141 141 139
28	DEC2	DEC2 Ws,Wd	DEC2 [W12++],[W11]--	Wd = Ws - 2	DEC2	140
29	DECSNZ	DECSNZ f	DECSNZ RAM100	f = f - 1, Skip if Not 0	DECSNZ	142
30	DECSZ	DECSZ f,Ww	DECSZ RAM100,Ww	Ww = f - 1, Skip if Not 0	DECSNZ	142
30	DECSZ	DECSZ f	DECSZ RAM100	f = f - 1, Skip if 0	DECSZ	143
30	DECSZ	DECSZ f,Ww	DECSZ RAM100,Ww	Ww = f - 1, Skip if 0	DECSZ	143
31	DISI	DISI lii14	DISI #157	Disable Interrupts for k instruction cycles	DISI	144

TABLE 1-1: DATA BOOK INSTRUCTION SET (CONTINUED)

Instr #	Assembly Mnemonic	Assembly Syntax	Example	Description	PLA Mnemonic	Page #
32	DIV		DIV	Divide Helper	DIV	145
33	DO	Sli16,li14	label-, #157	Do code to PC+Sli16, li14 times	DO	146
	DO	Sli16,Wn	label-,W3	Do code to PC+Sli16, (Wn) times	DOW	147
34	ED	A,Wm*Wm,Wxp,Wx,Wy	A,W2*W2,W0,[W5]+4,[W7]-2	Euclidean Distance	ED	148
35	EDAC	A,Wm*Wm,Wxp,Wx,Wy,AWB	A,W2*W2,W0,[W5]+4,[W7]-2,[W9]++	Euclidean Distance Accumulate	EDAC	150
36	EXCH	Wns,Wnd	W12,W11	Swap Wns with Wnd	EXCH	152
37	FBCL	Ws,Wd	[W12++],[W11]--	Find Bit Change from Left (MSb) Side	FBCL	153
38	FBCR	Ws,Wd	[W12++],[W11]--	Find Bit Change from Right (LSb) Side	FBCR	154
39	FFOL	Ws,Wd	[W12++],[W11]--	Find First Zero from Left (MSb) Side	FFOL	155
40	FFOR	Ws,Wd	[W12++],[W11]--	Find First Zero from Right (LSb) Side	FFOR	156
41	FF1L	Ws,Wd	[W12++],[W11]--	Find First One from Left (MSb) Side	FF1L	157
42	FF1R	Ws,Wd	[W12++],[W11]--	Find First One from Right (LSb) Side	FF1R	158
43	GOTO	li23	label	Go to address	GOTO	159
	GOTO	Wn	W11	Go to indirect	GOTOW	160
44	HALT			No Operation/ HALT	HALT	161
45	INC	f	RAM100	f = f + 1	INCF	164
	INC	f,Ww	RAM100,Ww	Ww = f + 1	INCF	164
	INC	Ws,Wd	[W12++],[W11]--	Wd = Ws + 1	INC	162
46	INC2	Ws,Wd	[W12++],[W11]--	Wd = Ws + 2	INC2	163
47	INCSNZ	f	RAM100	f = f + 1, Skip if Not 0	INCSNZ	165
	INCSNZ	f,Ww	RAM100,Ww	Ww = f + 1, Skip Not if 0	INCSNZ	165
48	INCSZ	f	RAM100	f = f + 1, Skip if 0	INCSZ	166
	INCSZ	f,Ww	RAM100,Ww	Ww = f + 1, Skip if 0	INCSZ	166
49	IOR	f	RAM100	f = f .IOR, Ww	IORWF	170
	IOR	f,Ww	RAM100,Ww	Ww = f .IOR, Ww	IORWF	170
	IOR	Sli10,Wn	#0xAA,W11	Wd = Sli10 .IOR, Wd	IORLW	169
	IOR	Wb,Ws,Wd	W7,[W12++],[W11]--	Wd = Wb .IOR, Ws	IOR	167
	IOR	Wb,li15,Wd	W7,#25,[W11]--	Wd = Wb .IOR, li15	IORLS	168
50	LAC	A,Wso,Sli14	A,[W12+6],#5	Load Accumulator	LAC	172
51	LNK	li14	#157	Link frame pointer	LNK	176
52	LSR	f	RAM100	f = Logical Right Shift f	LSRF	178
	LSR	f,Ww	RAM100,Ww	Ww = Logical Right Shift f	LSRF	178
	LSR	Ws,Wd	[W12++],[W11]--	Wd = Logical Right Shift Ws	LSR	177
	LSR	Wd,Wns,Wnd	W0,W2,W1	Wnd = Logical Right Shift Wb by Wns	LSRW	180
	LSR	Wd,li15,Wnd	W0,#23,W1	Wnd = Logical Right Shift Ws by li15	LSRK	179
53	MAC	A,Wm*Wm,Wxp,Wx,Wy,AWB	A,W2*W3,W0,[W5]+4,W1,[W7]-2,W9	Multiply and Accumulate	MAC	181
	MAC	A,Wm*Wm,Wxp,Wx,Wy,AWB	A,W2*W2,W0,[W5]+4,W1,[W7]-2,W9	Square and Accumulate	SQRAC	244

TABLE 1-1: DATA BOOK INSTRUCTION SET (CONTINUED)

Instr #	Assembly Mnemonic	Assembly Syntax	Example	Description	PLA Mnemonic	Page #
54	MOV	MOV f,Wn	MOV RAM100,W12	Move f to Wn	LDW	174
		MOV f	MOV RAM100	Move f to f	MOVF	184
		MOV f,Ww	MOV RAM100,Ww	Move f to Ww	MOVF	184
		MOV lit16,Wn	MOV #0x5A5A,W11	Move 16-bit literal to Wn	MOVL	185
		MOV Slit10,Wn	MOV #0xAA,W11	Move 10-bit signed literal to Wn	MOVLW	186
		MOV Wn,f	MOV W12,RAM100	Move Wn to f	STW	249
		MOV Wso,Wdo	MOV [W12+W3],[W11++]	Move Ws to Wd	MOV	183
		MOV Ww,f	MOV Ww,RAM100	Move Ww to f	MOVWF	189
		MOV.D Wns,Wd	MOV.D W12,[W11]--	Move W(ns):W(ns+1) to Wd	STDW	247
		MOV.Q Wns,Wd	MOV.Q W12,[W11]--	Move W(ns):W(ns+1):W(ns+2):W(ns+3) to Wd	STQW	248
55	MOV.SAC	MOV.SAC A,Wxp,Wx,Wyp,Wy,AWB	MOV.Q [W14++],W12	Move Ws to W(nd+1):W(nd)	LDDW	174
		MOV.SAC B,W3,[W5],W2,[W7],[W9]++	MOV.Q [W14++],W12	Move Ws to W(nd+3):W(nd+2):W(nd+1):W(nd)	LDDW	175
		MPY A,Wm*Wn,Wxp,Wx,Wyp,Wy	MPY A,W0*W1,W0,[W4]++=4	Move Special	MOVSAC	187
		MPY A,Wm*Wm,Wxp,Wx,Wyp,Wy	MPY A,W1*W1,W0,[W4]++=4	Multiply Wm by Wn to Accumulator	MPY	190
		MPYN A,Wm*Wn,Wxp,Wx,Wyp,Wy	MPYN B,W1*W2,W1,[W4],W2,[W6]++=2	Square Wm to Accumulator	SQR	242
		MSL Wb,Wns,Wnd	MSL W0,W2,W1	-(Multiply Wm by Wn) to Accumulator	MPYN	192
		MSL Wb,lit5,Wnd	MSL W0,#23,W1	Wnd = Multi-word Left Shift Wb by Wns	MSLW	197
		MSR Wb,Wns,Wnd	MSR W0,W2,W1	Wnd = Multi-word Left Shift Wb by lit5	MSLK	196
		MSR Wb,lit5,Wnd	MSR W0,#23,W1	Wnd = Multi-word Right Shift Wb by Wns	MSRW	199
		MSC A,Wm*Wn,Wxp,Wx,Wyp,Wy,AWB	MSC A,W2*W3,W0,[W5]++=4,W1,[W6],W9	Wnd = Multi-word Right Shift Wb by lit5	MSRK	198
61	MUL	MUL.SS Wb,Ws,Wnd	MUL.SS W7,[W12++],W11	Multiply and Subtract from Accumulator	MSC	194
		MUL.SU Wb,Ws,Wnd	MUL.SU W7,[W12++],W11	[Wd+1, Wd] = signed(Wb) * signed(Ws)	MULS	200
		MUL.US Wb,Ws,Wnd	MUL.US W7,[W12++],W11	[Wd+1, Wd] = signed(Wb) * unsigned(Ws)	MULSU	201
		MUL.UU Wb,Ws,Wnd	MUL.UU W7,[W12++],W11	[Wd+1, Wd] = unsigned(Wb) * signed(Ws)	MULUS	205
		MUL.SU Wb,lit5,Wnd	MUL.SU W7,#25,W11	[Wd+1, Wd] = unsigned(Wb) * unsigned(Ws)	MULU	203
		MUL.UU Wb,lit5,Wnd	MUL.UU W7,#25,W11	[Wd+1, Wd] = signed(Wb) * unsigned(lit5)	MUL.SULS	202
		MUL f	MUL RAM100	[Wd+1, Wd] = unsigned(Wb) * unsigned(lit5)	MULULS	204
		NEG A	NEG B	W3:W2 = f * Ww	MULWF	206
		NEG f	NEG RAM100	Negate Accumulator	NEGAB	208
		NEG f,Ww	NEG RAM100,Ww	f = f + 1	NEGF	209
63	NOP	NEG Ws,Wd	NEG [W12++],[W11]--	Ww = f + 1	NEGF	209
		NOP	NOP	Wd = Ws + 1	NEG	207
		NOPR	NOPR	No Operation	NOP	210
		POP f	POP RAM100	No Operation	NOPR	211
		POP.S	POP.S	Pop f from top of stack (TOS)	POP	212
		POP Wdo	POP [W11+6]	Pop Shadow Registers	ITCH	171
		POP.D Wnd	POP.D W12	Pop Wd Registers	MOV	183
		POP.Q Wnd	POP.Q W12	Pop W(nd+1):W(nd) Registers	LDDW	174
				Pop W(nd+3):W(nd+2):W(nd+1):W(nd) Registers	LDDW	175

TABLE 1-1: DATA BOOK INSTRUCTION SET (CONTINUED)

Instr #	Assembly Mnemonic	Assembly Syntax	Example	Description	PLA Mnemonic	Page #
65	PUSH	PUSH f	PUSH RAM100	Push f to top of stack (TOS)	PUSH	213
		PUSH.S	PUSH.S	Push Shadow Registers	SCRATCH	231
		PUSH Wso	PUSH [W12+W3]	Push Ws Registers	MOV	183
		PUSH.D Wns	PUSH.D W12	Push W(ns):W(ns+1) Registers	STDW	247
		PUSH.Q Wns	PUSH.Q W12	Push W(ns):W(ns+1):W(ns+2):W(ns+3) Registers	STQW	248
66	RCALL	RCALL Slii16	RCALL label	Relative Call	RCALL	215
		RCALL Wn	RCALL W11	Computed Call	RCALLW	215
67	REPEAT	REPEAT liti14	REPEAT #157	Repeat Next Instruction liti14 times	REPEAT	216
		REPEAT Wn	REPEAT W11	Repeat Next Instruction (Wn) times	REPEATW	217
68	RESET	RESET	RESET	Software device RESET	RESET	218
69	RETFIE	RETFIE	RETFIE	Return from interrupt enable	RETFIE	219
		RETFIE.S	RETFIE.S			
70	RETLW	RETLW Slii10,Wn	RETLW #0xAA,W11	Return with literal in Wn	RETLW	220
71	RETURN	RETURN	RETURN	Return from Subroutine	RETURN	221
		RETURN.S	RETURN.S			
72	RLC	RLC f	RLC RAM100	f = Rotate Left through Carry f	RLCF	223
		RLC f,Ww	RLC RAM100,Ww	Ww = Rotate Left through Carry f	RLCF	223
		RLC Ws,Wd	RLC [W12++],[W11]--	Wd = Rotate Left through Carry Ws	RLC	222
73	RLNC	RLNC f	RLNC RAM100	f = Rotate Left (No Carry) f	RLNCF	225
		RLNC f,Ww	RLNC RAM100,Ww	Ww = Rotate Left (No Carry) f	RLNCF	225
		RLNC Ws,Wd	RLNC [W12++],[W11]--	Wd = Rotate Left (No Carry) Ws	RLNC	224
74	RRC	RRC f	RRC RAM100	f = Rotate Right through Carry f	RRCF	227
		RRC f,Ww	RRC RAM100,Ww	Ww = Rotate Right through Carry f	RRCF	227
		RRC Ws,Wd	RRC [W12++],[W11]--	Wd = Rotate Right through Carry Ws	RRC	226
75	RRNC	RRNC f	RRNC RAM100	f = Rotate Right (No Carry) f	RRNCF	229
		RRNC f,Ww	RRNC RAM100,Ww	Ww = Rotate Right (No Carry) f	RRNCF	229
		RRNC Ws,Wd	RRNC [W12++],[W11]--	Wd = Rotate Right (No Carry) Ws	RRNC	228
76	SAC	SAC A,Wdo,Slii4	SAC A,[W11+W3],#5	Store Accumulator	SAC	230
		SAC.R A,Wdo,Slii4	SAC.R A,[W11+W3],#5	Store Rounded Accumulator	SRAC	246
77	SE	SE Ws,Wd	SE [W12++],[W11]--	Wd = sign extended Ws	SE	232
78	SETM	SETM f	SETM RAM100	f = 0xFFFF	SETF	234
		SETM Ww	SETM Ww	Ww = 0xFFFF	SETF	234
		SETM Ws	SETM [W11]--	Ws = 0xFFFF	SETM	233
79	SFTAC	SFTAC A,Wn	SFTAC A,W12	Arithmetic Shift by (Wn) Accumulator	SFTAC	236
		SFTAC A,Slii5	SFTAC A,#5	Arithmetic Shift by Slii5 Accumulator	SFTACK	236

TABLE 1-1: DATA BOOK INSTRUCTION SET (CONTINUED)

Instr #	Assembly Mnemonic	Assembly Syntax	Example	Description	PLA Mnemonic	Page #
80	SL	f	SL RAM100	f = Left Shift f	SLF	239
		f,Ww	SL RAM100,Ww	Ww = Left Shift f	SLF	239
		Ws,Wd	SL [W12++],[W11]--	Wd = Left Shift Ws	SL	237
		Wd,Wns,Wnd	SL W0,W2,W1	Wd = Left Shift Wb by Wns	SLW	241
		Wd,lit5,Wnd	SL W0,#23,W1	Wd = Left Shift Ws by lit5	SLK	240
81	SLEEP	lit4	SLEEP #5	Go into standby mode	SLEEP	238
82	SUB	A	SUB B	Subtract Accumulators	SUBAB	251
		f	SUB RAM100	f = f - Ww	SUBWF	264
		f,Ww	SUB RAM100,Ww	Ww = f - Ww	SUBWF	264
		Slit10,Wn	SUB #0xAA,W11	Wd = Slit10 - Wd	SUBLW	261
		Wb,Ws,Wd	SUB W7,[W12++],[W11]--	Wd = Wb - Ws	SUB	250
83	SUBB	Wb,lit5,Wd	SUB W7,#25,[W11]--	Wd = Wb - lit5	SUBLS	260
		f	SUBB RAM100	f = f - Ww - (C)	SUBBWF	258
		f,Ww	SUBB RAM100,Ww	Ww = f - Ww - (C)	SUBBWF	258
		Slit10,Wn	SUBB #0xAA,W11	Wd = Slit10 - Wd - (C)	SUBBLW	255
		Wb,Ws,Wd	SUBB W7,[W12++],[W11]--	Wd = Wb - Ws - (C)	SUBB	252
84	SUBR	Wb,lit5,Wd	SUBB W7,#25,[W11]--	Wd = Wb - lit5 - (C)	SUBBLS	254
		f	SUBR RAM100	f = Ww - f	SUBFW	264
		f,Ww	SUBR RAM100,Ww	Ww = Ww - f	SUBFW	264
		Wb,Ws,Wd	SUBR W7,[W12++],[W11]--	Wd = Wb - Ws	SUBR	262
		Wb,lit5,Wd	SUBR W7,#25,[W11]--	Wd = Wb - lit5	SUBRLS	263
85	SUBBR	f	SUBBR RAM100	f = Ww - f - (C)	SUBBFW	258
		f,Ww	SUBBR RAM100,Ww	Ww = Ww - f - (C)	SUBBFW	258
		Wb,Ws,Wd	SUBBR W7,[W12++],[W11]--	Wd = Ws - Wb - (C)	SUBBR	256
		Wb,lit5,Wd	SUBBR W7,#25,[W11]--	Wd = lit5 - Wb - (C)	SUBBRBS	257
		Wn	SWAP.b W11	Wn = nibble swap Wn	SWAP	265
86	SWAP	Wn	SWAP W11	Wn = byte swap Wn	SWAP	265
87	TBLRDH	Ws,Wd	TBLRDH [W12++],[W11]--	Read Prog<23:16> to Wd	TBLRDH	266
88	TBLRDL	Ws,Wd	TBLRDL [W12++],[W11]--	Read Prog<15:0> to Wd	TBLRDL	268
89	TBLWTH	Ws,Wd	TBLWTH [W12++],[W11]--	Write Ws to Prog<23:16>	TBLWTH	270
90	TBLWTL	Ws,Wd	TBLWTL [W12++],[W11]--	Write Ws to Prog<15:0>	TBLWTL	272
91	TRAP	lit1,lit16	TRAP 0,#157	Trap to vector with literal	TRAP	275
92	ULNK		ULNK	Unlink frame pointer	ULNK	274
93	XOR	f	XOR RAM100	f = f .XOR. Ww	XORWF	279
	XOR	f,Ww	XOR RAM100,Ww	Ww = f .XOR. Ww	XORWF	279
		Slit10,Wn	XOR #0xAA,W11	Wd = Slit10 .XOR. Wd	XORLW	278
		Wb,Ws,Wd	XOR W7,[W12++],[W11]--	Wd = Wb .XOR. Ws	XOR	276
		Wb,lit5,Wd	XOR W7,#25,[W11]--	Wd = Wb .XOR. lit5	XORLS	277
		Ws,Wd	ZE [W12++],[W11]--	Wd = Zero Extend Ws	ZE	





**TABLE 1-2: ROADRUNNER INSTRUCTION SET CODING (CONTINUED)**

PLA Mnemonic	Assembly Syntax Mnemonic, Operands	Description	W	CY	Opcode																Note	Page #								
Math Operations - W Registers																														
ADD	ADD Wb,Ws,Wd	Wd = Wb + Ws	1	1	0	1	0	0	0	w	w	w	w	w	B	q	q	d	d	d	d	p	p	s	s	s	1	61		
ADDC	ADDC Wb,Ws,Wd	Wd = Wb + Ws + (C)	1	1	0	1	0	0	1	w	w	w	w	w	B	q	q	d	d	d	d	p	p	p	s	s	s	1	64	
AND	AND Wb,Ws,Wd	Wd = Wb .AND. Ws	1	1	0	1	1	0	0	w	w	w	w	w	B	q	q	d	d	d	d	p	p	p	s	s	s	1	71	
IOR	IOR Wb,Ws,Wd	Wd = Wb .IOR. Ws	1	1	0	1	1	0	0	w	w	w	w	w	B	q	q	d	d	d	d	p	p	p	s	s	s	1	167	
SUB	SUB Wb,Ws,Wd	Wd = Wb - Ws	1	1	0	1	0	1	0	w	w	w	w	w	B	q	q	d	d	d	d	p	p	p	s	s	s	1	250	
SUBB	SUBB Wb,Ws,Wd	Wd = Wb - Ws - (C)	1	1	0	1	0	1	1	w	w	w	w	w	B	q	q	d	d	d	d	p	p	p	s	s	s	1	252	
SUBR	SUBR Wb,Ws,Wd	Wd = Ws - Wb	1	1	0	0	1	0	0	w	w	w	w	w	B	q	q	d	d	d	d	p	p	p	s	s	s	1	262	
SUBBR	SUBBR Wb,Ws,Wd	Wd = Ws - Wb - (C)	1	1	0	0	1	1	0	w	w	w	w	w	B	q	q	d	d	d	d	p	p	p	s	s	s	1	256	
XOR	XOR Wb,Ws,Wd	Wd = Wb .XOR. Ws	1	1	0	1	1	0	1	w	w	w	w	w	B	q	q	d	d	d	d	p	p	p	s	s	s	1	276	
Math Operations - Short Literals (literal 0..31)																														
ADDLS	ADD Wb,lii5,Wd	Wd = Wb + lii5	1	1	0	1	0	0	0	w	w	w	w	w	B	q	q	d	d	d	d	1	1	k	k	k	k	1	67	
ADDCLS	ADDC Wb,lii5,Wd	Wd = Wb + lii5 + (C)	1	1	0	1	0	0	1	w	w	w	w	w	B	q	q	d	d	d	d	1	1	k	k	k	k	1	65	
ANDLS	AND Wb,lii5,Wd	Wd = Wb .AND. lii5	1	1	0	1	1	0	0	w	w	w	w	w	B	q	q	d	d	d	d	1	1	k	k	k	k	1	72	
IORLS	IOR Wb,lii5,Wd	Wd = Wb .IOR. lii5	1	1	0	1	1	0	0	w	w	w	w	w	B	q	q	d	d	d	d	1	1	k	k	k	k	1	168	
SUBLS	SUB Wb,lii5,Wd	Wd = Wb - lii5	1	1	0	1	0	1	0	w	w	w	w	w	B	q	q	d	d	d	d	1	1	k	k	k	k	1	260	
SUBBLS	SUBB Wb,lii5,Wd	Wd = Wb - lii5 - (C)	1	1	0	1	0	1	1	w	w	w	w	w	B	q	q	d	d	d	d	1	1	k	k	k	k	1	254	
SUBRLS	SUBR Wb,lii5,Wd	Wd = lii5 - Wb	1	1	0	0	1	0	0	w	w	w	w	w	B	q	q	d	d	d	d	1	1	k	k	k	k	1	263	
SUBBRLS	SUBBR Wb,lii5,Wd	Wd = lii5 - Wb - (C)	1	1	0	0	1	1	0	w	w	w	w	w	B	q	q	d	d	d	d	1	1	k	k	k	k	1	257	
XORLS	XOR Wb,lii5,Wd	Wd = Wb .XOR. lii5	1	1	0	1	1	0	1	w	w	w	w	w	B	q	q	d	d	d	d	1	1	k	k	k	k	1	277	
Math Operations - W Registers Single Operand																														
CLR	CLR Ws	Ws = 0x0000	1	1	1	1	0	1	0	1	1	0	1	1	0	B	0	0	0	0	0	0	p	p	p	s	s	s	1	117
COM	COM Ws,Wd	Wd = Ws	1	1	1	1	0	1	0	1	0	1	0	1	1	B	q	q	d	d	d	d	p	p	p	s	s	s	1	122
DEC	DEC Ws,Wd	Wd = Ws - 1	1	1	1	1	0	1	0	0	1	0	1	0	1	B	q	q	d	d	d	d	p	p	p	s	s	s	1	139
DEC2	DEC2 Ws,Wd	Wd = Ws - 2	1	1	1	1	0	1	0	0	1	0	1	1	B	q	q	d	d	d	d	p	p	p	s	s	s	1	140	
INC	INC Ws,Wd	Wd = Ws + 1	1	1	1	1	0	1	0	0	0	0	0	0	B	q	q	d	d	d	d	p	p	p	s	s	s	1	162	
INC2	INC2 Ws,Wd	Wd = Ws + 2	1	1	1	1	0	1	0	0	0	1	0	0	1	B	q	q	d	d	d	d	p	p	p	s	s	s	1	163
NEG	NEG Ws,Wd	Wd = $\overline{Ws}$ + 1	1	1	1	1	0	1	0	1	0	1	0	0	B	q	q	d	d	d	d	p	p	p	s	s	s	1	207	
SETM	SETM Ws	Ws = 0xFFFF	1	1	1	1	0	1	0	1	1	1	1	1	B	0	0	0	0	0	0	0	p	p	p	s	s	s	1	233

Note 1: INST or INSTW is a word operation,B=0; INST.B is a byte operation,B=1.

PLA Mnemonic	Assembly Syntax Mnemonic, Operands	Description	W	CY	Opcode																Note	Page #
					2	2	2	2	1	1	1	1	1	1	1	1	0	0	0	0	0	0
					3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6
<b>Math Operations - File Registers</b>																						
ADDWF	ADD f f, Ww	destination = f + Ww	1	1	1	0	1	1	0	1	0	0	0	B	D	f	f	f	f	f	f	f
ADDWFC	ADDC f f, Ww	destination = f + Ww + (C)	1	1	1	0	1	1	0	1	0	0	1	B	D	f	f	f	f	f	f	f
ANDWF	AND f f, Ww	destination = f .AND. Ww	1	1	1	0	1	1	0	1	0	0	0	B	D	f	f	f	f	f	f	f
IORWF	IOR f f, Ww	destination = f .IOR. Ww	1	1	1	0	1	1	1	1	0	1	0	B	D	f	f	f	f	f	f	f
SUBWF	SUBR f f, Ww	destination = Ww - f	1	1	1	0	1	1	1	0	1	0	0	B	D	f	f	f	f	f	f	f
SUBBFW	SUBRB f f, Ww	destination = Ww - f - (C)	1	1	1	0	1	1	1	0	1	1	B	D	f	f	f	f	f	f	f	f
SUBWF	SUB f f, Ww	destination = f - Ww	1	1	1	0	1	1	0	1	0	1	0	B	D	f	f	f	f	f	f	f
SUBBWF	SUBB f f, Ww	destination = f - Ww - (C)	1	1	1	0	1	1	0	1	1	1	B	D	f	f	f	f	f	f	f	f
XORWF	XOR f f, Ww	destination = f .XOR. Ww	1	1	1	0	1	1	0	1	1	0	1	B	D	f	f	f	f	f	f	f
<b>Math Operations - File Registers Single Operand</b>																						
CLRF	CLR f Ww	destination = 0x0000	1	1	1	1	0	1	1	1	1	0	0	B	D	f	f	f	f	f	f	f
COMF	COM f f, Ww	destination = $\bar{f}$	1	1	1	1	0	1	1	1	0	1	0	B	D	f	f	f	f	f	f	f
DECf	DEC f f, Ww	destination = f - 1	1	1	1	1	0	1	1	0	1	0	1	B	D	f	f	f	f	f	f	f
INCF	INC f f, Ww	destination = f + 1	1	1	1	1	0	1	1	0	1	0	0	B	D	f	f	f	f	f	f	f
NEGF	NEG f f, Ww	destination = $\bar{f} + 1$	1	1	1	1	0	1	1	1	0	0	B	D	f	f	f	f	f	f	f	f
SETF	SETM f Ww	destination = 0xFFFF	1	1	1	1	0	1	1	1	1	1	1	B	D	f	f	f	f	f	f	f

Note 1: INST or INST.W is a word operation, B=0; INST.B is a byte operation, B=1.

Note 2: Destination is Ww if D=0; f if D=1.

PLA Mnemonic	Assembly Syntax Mnemonic, Operands	Description	W	CY	Opcode																Note	Page #						
					3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
Math Operations - Literals (literal -512..511)																												
ADDLW	ADD Sllt10, Wn	Wn = Sllt10 + Wn	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	68
ADDCLW	ADDC Sllt10, Wn	Wn = Sllt10 + Wn + (C)	1	1	1	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	66
ANDLW	AND Sllt10, Wn	Wn = Sllt10 .AND. Wn	1	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	73
IORLW	IOR Sllt10, Wn	Wn = Sllt10 .IOR. Wn	1	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	169
SUBLW	SUB Sllt10, Wn	Wn = Sllt10 - Wn	1	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	261
SUBBLW	Sllt10, Wn	Wn = Sllt10 - Wn - (C)	1	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	255
XORLW	XOR Sllt10, Wn	Wn = Sllt10 .XOR. Wn	1	1	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	278
Math Operations - Multiply, Adjust																												
DAW	DAW.B Wn	Wn = decimal adjust Wn	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	138
DIV	DIV	Divide Helper	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	145
MULS	MUL.SS Wb, Ws, Wnd	{Wnd+1, Wnd} = sign(Wb) * sign(Ws)	1	1	1	0	1	1	0	0	1	1	w	w	w	d	d	d	d	d	d	p	p	p	s	s	s	200
MULSU	MUL.SU Wb, Ws, Wnd	{Wnd+1, Wnd} = sign(Wb) * unsign(Ws)	1	1	1	0	1	1	0	0	1	0	w	w	w	d	d	d	d	d	d	p	p	p	s	s	s	201
MULSLS	MUL.SU Wb, lit5, Wnd	{Wnd+1, Wnd} = sign(Wb) * unsign(lit5)	1	1	1	0	1	1	0	0	1	0	w	w	w	d	d	d	d	d	d	1	1	k	k	k	k	202
MULU	MUL.UU Wb, Ws, Wnd	{Wnd+1, Wnd} = unsign(Wb) * unsign(Ws)	1	1	1	0	1	1	0	0	0	0	w	w	w	d	d	d	d	d	d	d	p	p	p	s	s	203
MULULS	MUL.UU Wb, lit5, Wnd	{Wnd+1, Wnd} = unsign(Wb) * unsign(lit5)	1	1	1	0	1	1	0	0	0	0	w	w	w	d	d	d	d	d	d	1	1	k	k	k	k	204
MULUS	MUL.US Wb, Ws, Wnd	{Wnd+1, Wnd} = unsign(Wb) * sign(Ws)	1	1	1	0	1	1	0	0	1	0	1	w	w	w	d	d	d	d	d	d	p	p	p	s	s	205
MULWF	MUL f	W3:W2 = f * Ww	1	1	1	1	1	1	0	0	0	0	1	w	w	w	d	d	d	d	d	d	p	p	p	s	s	206
SE	SE Ws, Wd	Wd = sign extended Ws	1	1	1	1	1	1	0	0	1	0	0	q	q	q	d	d	d	d	d	p	p	p	s	s	s	292
ZE	ZE Ws, Wd	Wd = zero extended Ws	1	1	1	1	1	1	0	1	1	0	0	q	q	q	d	d	d	d	d	p	p	p	s	s	s	
SWAP	SWAP Wn	Wn = byte or nibble swap Wn	1	1	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	265

Note 1: INST or INST.W is a word operation, B=0; INST.B is a byte operation, B=1.



TABLE 1-2: ROADRUNNER INSTRUCTION SET CODING (CONTINUED)

PLA Mnemonic	Assembly Syntax Mnemonic, Operands	Description	W	CY	Opcode														Note	Page #
2 2																				

**TABLE 1-2: ROADRUNNER INSTRUCTION SET CODING (CONTINUED)**

PLA Mnemonic	Assembly Syntax Mnemonic, Operands	Description	W	CY	Opcode																Note	Page #
					2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0	0	0
					3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6
<b>DSP OPERATIONS - Accumulator Ops</b>																						
ADDAB	ADD A	Add Accumulators	1	1	1	1	0	0	1	0	1	1	A	0	0	0	0	0	0	0	0	0
ADDAC	ADD A,Wso,Sli4	16-bit Signed Add to Accumulator	1	1	1	1	0	0	1	0	0	1	A	w	w	r	r	r	r	r	r	r
LAC	LAC A,Wso,Sli4	Load Accumulator	1	1	1	1	0	0	1	0	1	0	A	w	w	r	r	r	r	r	r	r
NEGAB	NEG A	Negate Accumulators	1	1	1	1	0	0	1	0	1	1	A	0	0	1	0	0	0	0	0	0
SAC	SAC A,Wdo,Sli4	Store Accumulator	1	1	1	1	0	0	1	1	0	0	A	w	w	w	r	r	r	r	r	r
SFTAC	SFTAC A,Wn	Arithmetic Shift by (Wn) Accumulator	1	1	1	1	0	0	1	0	0	0	A	0	0	0	0	0	0	0	s	s
SFTACK	SFTACK A,Sli5	Arithmetic Shift by Sli5 Accumulator	1	1	1	1	0	0	1	0	0	0	A	0	0	0	0	0	0	1	k	k
SRAC	SAC.R A,Wdo,Sli4	Store Rounded Accumulator	1	1	1	1	0	0	1	1	0	1	A	w	w	w	r	r	r	r	h	h
SUBAB	SUB A	Subtract Accumulators	1	1	1	1	0	0	1	0	1	1	A	0	1	1	0	0	0	0	0	0
<b>DSP OPERATIONS - MAC Ops</b>																						
CLRAC	CLR A,Wxp,Wx,Wyp,Wy,AWB	Clear Accumulator	1	1	1	1	0	0	0	0	1	1	A	0	x	x	y	y	i	i	j	j
ED	ED A,Wm*Wm,Wxp,Wx,Wy	Euclidean Distance	1	1	1	1	1	0	0	m	m	m	A	1	x	x	0	0	i	i	j	j
EDAC	EDAC A,Wm*Wm,Wxp,Wx,Wy,AWB	Euclidean Distance Accumulate	1	1	1	1	1	0	0	m	m	m	A	1	x	x	0	0	i	i	j	j
MAC	MAC A,Wm*Wn,Wxp,Wx,Wyp,Wy,AWB	Multiply and Accumulate	1	1	1	1	0	0	0	m	m	m	A	0	x	x	y	y	i	i	j	j
MOVSAc	MOVSAc A,Wxp,Wx,Wyp,Wy,AWB	Move Special	1	1	1	1	0	0	0	1	1	1	A	0	x	x	y	y	i	i	j	j
MPY	MPY A,Wm*Wn,Wxp,Wx,Wyp,Wy	Multiply Wn by Wm to Accumulator	1	1	1	1	0	0	0	m	m	m	A	0	x	x	y	y	i	i	j	j
MPYN	MPYN A,Wm*Wn,Wxp,Wx,Wyp,Wy	-(Multiply Wn by Wm) to Accumulator	1	1	1	1	0	0	0	m	m	m	A	1	x	x	y	y	i	i	j	j
MSC	MSC A,Wm*Wn,Wxp,Wx,Wyp,Wy,AWB	Multiply and Subtract from Accumulator	1	1	1	1	0	0	0	m	m	m	A	1	x	x	y	y	i	i	j	j
SQR	SQR A,Wm*Wm,Wxp,Wx,Wyp,Wy	Square to Accumulator	1	1	1	1	1	0	0	m	m	m	A	0	x	x	y	y	i	i	j	j
SQRAc	SQRAc A,Wm*Wm,Wxp,Wx,Wyp,Wy,AWB	Square and Accumulate	1	1	1	1	1	0	0	m	m	m	A	0	x	x	y	y	i	i	j	j

Note 5: lit4 translates to the rrrr field that specifies a shift count

TABLE 1-2: ROADRUNNER INSTRUCTION SET CODING (CONTINUED)

PLA Mnemonic	Assembly Syntax Mnemonic, Operands	Description	W	CY	Opcode																Note	Page #																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																										
					2	2	2	2	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Note 1: INST or INST.W is a word operation, B=0; INST.B is a byte operation, B=1.

Note 3: bbbb field selects bit position 1111=MSb(15) 0000=LSb(0)

Note 4: bbb field selects bit position 111=MSb(7) 000=LSb(0)





PLA Mnemonic	Assembly Syntax Mnemonic, Operands	Description	W	CY	Opcode																Note	Page #																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																															
					2	2	2	2	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



TABLE 1-2: ROADRUNNER INSTRUCTION SET CODING (CONTINUED)

PLA Mnemonic	Assembly Syntax Mnemonic, Operands	Description	W	CY	Opcode																Note	Page #											
					2	2	2	2	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0		
					3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0					
Jump / Call / Return Operations																																	
BRAW	BRA Wn	Computed branch	1	2	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		97
CALL	CALL lit23	Call subroutine	2	2	0	0	0	0	0	0	1	S	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	7	115
	CALL.S				0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
CALLW	CALL Wn	Call indirect subroutine	1	2	0	0	0	0	0	0	0	1	S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		116
	CALL.S				0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
GOTO	GOTO lit23	Go to address	2	2	0	0	0	0	0	1	0	0	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	7	159
					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
GOTOW	GOTO Wn	Go to indirect	1	2	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		160
RCALL	RCALL Slit16	Relative Call	1	2	0	0	0	0	1	1	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	6	215
RCALLW	RCALL Wn	Computed Call	1	2	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		215
RETIE	RETIE RETIE.S	Return from interrupt enable	1	2	0	0	0	0	1	1	0	S	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		219
RETLW	RETLW Slit10,Wn	Return with Slit10 in Wn	1	2	0	0	0	0	1	0	1	S	B	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	1	220	
	RETLW.S				0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
RETURN	RETURN RETURN.S	Return from Subroutine	1	2	0	0	0	0	1	1	0	S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		221
					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
TRAP	TRAP lit1,lit16	Trap to vector(lit1) with lit16	1	2	0	0	0	0	1	0	1	n	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k		275
Looping Operations																																	
DO	DO Slit16,lit14	Do code to PC+Slit16, lit14 times	2	2+ loop	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	146
					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
DOW	DO Slit16,Wn	Do code to PC+Slit16, (Wn) times	2	2+ loop	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	147
					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
REPEAT	REPEAT lit14	Repeat Next Instruction lit14 times	1	1 + lit14	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		216
					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
REPEATW	REPEAT Wn	Repeat Next Instruction (Wn) times	1	1 + (Wn)	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		217
					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
Note 1: INST or INST.W is a word operation,B=0; INST.B is a byte operation,B=1.																																	
Note 6: 16-bit literal nnnnnnnnnnnnnnnnn allows jump range of PC-32768 to PC+32767																																	
Note 7: 23-bit literal coded in 2 words, 1st word contains n<15:0>, n<0>=>0, 2nd word contains n<22:16>.																																	

Note 1: INST or INST.W is a word operation, B=0; INST.B is a byte operation, B=1.

Note 6: 16-bit literal mmmmmmmmmmmmmmmmm allows jump range of PC-32768 to PC+32767

Note 7: 23-bit literal coded in 2 words, 1st word contains n&lt;15:0&gt;, n&lt;0&gt;=&gt;0, 2nd word contains n&lt;22:16&gt;.

TABLE 1-2: ROADRUNNER INSTRUCTION SET CODING (CONTINUED)

PLA Mnemonic	Assembly Syntax Mnemonic, Operands	Description	W	CY	Opcode																Note	Page #
					2	2	2	2	1	1	1	1	1	1	1	1	1	1	0	0	0	0
					3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6
					5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0		
<b>Stack Operations</b>																						
ITCH	POPS	Pop Shadow Registers	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
LNK	LNK lit14	Link frame pointer	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
POP	POP f	Pop f from top of stack (TOS)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
PUSH	PUSH f	Push f to top of stack (TOS)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
SCRATCH	PUSH.S	Push Shadow Registers	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
ULNK	ULNK	Unlink frame pointer	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
<b>Control Operations</b>																						
CLRWDT	CLRWDT	Clear Watchdog Timer	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
DISI	DISI lit14	Disable Interrupts for lit14 instruction cycles	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
HALT	HALT	No Operation/ HALT	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
NOP	NOP	No Operation	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
NOPR	NOPR	No Operation	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
RESET	RESET	Software device RESET	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
SLEEP	SLEEP lit4	Go into standby mode	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0

TABLE 1-3: ROADRUNNER STATUS FLAG OPERATIONS

PLA Mnemonic	Status Affected	C	DC	N	OV	SZ	Z	OA	OB	SA	SB
<b>Move Operations</b>											
EXCH	None	—	—	—	—	—	—	—	—	—	—
LDDW	None	—	—	—	—	—	—	—	—	—	—
LDQW	None	—	—	—	—	—	—	—	—	—	—
LDW	None	—	—	—	—	—	—	—	—	—	—
MOV	None	—	—	—	—	—	—	—	—	—	—
MOVF	N,Z	—	—	↕	—	—	↕	—	—	—	—
MOVL	None	—	—	—	—	—	—	—	—	—	—
MOVLW	None	—	—	—	—	—	—	—	—	—	—
MOVWF	None	—	—	—	—	—	—	—	—	—	—
STDW	None	—	—	—	—	—	—	—	—	—	—
STQW	None	—	—	—	—	—	—	—	—	—	—
STW	None	—	—	—	—	—	—	—	—	—	—
<b>Table Operations</b>											
TBLRDH	None	—	—	—	—	—	—	—	—	—	—
TBLRDL	None	—	—	—	—	—	—	—	—	—	—
TBLWTH	None	—	—	—	—	—	—	—	—	—	—
TBLWTL	None	—	—	—	—	—	—	—	—	—	—
<b>Math Operations - W Registers</b>											
ADD	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	↕	↕
ADDC	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
AND	N,SZ,Z	—	—	↕	—	↕	↕	—	—	—	—
IOR	N,SZ,Z	—	—	↕	—	↕	↕	—	—	—	—
SUB	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
SUBB	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
SUBR	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
SUBBR	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
XOR	N,SZ,Z	—	—	↕	—	↕	↕	—	—	—	—
<b>Math Operations - Short Literals (literal 0..31)</b>											
ADDLS	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
ADDCLS	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
ANDLS	N,SZ,Z	—	—	↕	—	↕	↕	—	—	—	—
IORLS	N,SZ,Z	—	—	↕	—	↕	↕	—	—	—	—
SUBLS	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
SUBBLS	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
SUBRLS	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
SUBBRLS	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
XORLS	N,SZ,Z	—	—	↕	—	↕	↕	—	—	—	—
<b>Math Operations - W Registers Single Operand</b>											
CLR	Z	—	—	—	—	—	1	—	—	—	—
COM	N,SZ,Z	—	—	↕	—	↕	↕	—	—	—	—
DEC	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
DEC2	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
INC	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
INC2	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
NEG	C,DC,N,OV,SZ,Z	↕	↕	↕	↕	↕	↕	—	—	—	—
SETM	None	—	—	—	—	—	—	—	—	—	—

TOTAL: 06034570459

TABLE 1-3: ROADRUNNER STATUS FLAG OPERATIONS (CONTINUED)

PLA Mnemonic	Status Affected	C	DC	N	OV	SZ	Z	OA	OB	SA	SB
<b>Math Operations - File Registers</b>											
ADDWF	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
ADDWFC	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
ANDWF	N,SZ,Z	—	—	0	—	0	0	—	—	—	—
IORWF	N,SZ,Z	—	—	0	—	0	0	—	—	—	—
SUBFW	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
SUBBFW	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
SUBWF	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
SUBBWF	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
XORWF	N,SZ,Z	—	—	0	—	0	0	—	—	—	—
<b>Math Operations - File Registers Single Operand</b>											
CLRF	Z	—	—	—	—	—	1	—	—	—	—
COMF	N,SZ,Z	—	—	0	—	0	0	—	—	—	—
DECF	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
INCF	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
NEGF	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
SETF	None	—	—	—	—	—	—	—	—	—	—
<b>Math Operations - Literals (literal -512..511)</b>											
ADDLW	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
ADDCLW	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
ANDLW	N,SZ,Z	—	—	0	—	0	0	—	—	—	—
IORLW	N,SZ,Z	—	—	0	—	0	0	—	—	—	—
SUBLW	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
SUBBLW	C,DC,N,OV,SZ,Z	0	0	0	0	0	0	—	—	—	—
XORLW	N,Z	—	—	0	—	0	0	—	—	—	—
<b>Math Operations - Multiply, Adjust</b>											
DAW	C	0	—	—	—	—	—	—	—	—	—
DIV	None	—	—	—	—	—	—	—	—	—	—
MULS	None	—	—	—	—	—	—	—	—	—	—
MULSU	None	—	—	—	—	—	—	—	—	—	—
MULSULS	None	—	—	—	—	—	—	—	—	—	—
MULU	None	—	—	—	—	—	—	—	—	—	—
MULULS	None	—	—	—	—	—	—	—	—	—	—
MULUS	None	—	—	—	—	—	—	—	—	—	—
MULWF	None	—	—	—	—	—	—	—	—	—	—
SE	C,N,Z	0	—	0	—	0	—	—	—	—	—
ZE	None	—	—	—	—	—	—	—	—	—	—
SWAP	None	—	—	—	—	—	—	—	—	—	—

T030457-060401

TABLE 1-3: ROADRUNNER STATUS FLAG OPERATIONS (CONTINUED)

PLA Mnemonic	Status Affected	C	DC	N	OV	SZ	Z	OA	OB	SA	SB
<b>Rotate/Shift Operations - W Registers</b>											
ASR	C,N,OV,SZ,Z	↻	—	↻	↻	↻	↻	—	—	—	—
LSR	C,N,OV,SZ,Z	↻	—	↻	↻	↻	↻	—	—	—	—
RLC	C,N,SZ,Z	↻	—	↻	—	↻	↻	—	—	—	—
RLNC	N,SZ,Z	—	—	↻	—	↻	↻	—	—	—	—
RRC	C,N,SZ,Z	↻	—	↻	—	↻	↻	—	—	—	—
RRNC	N,SZ,Z	—	—	↻	—	↻	↻	—	—	—	—
SL	C,N,OV,SZ,Z	↻	—	↻	↻	↻	↻	—	—	—	—
<b>Rotate/Shift Operations - File Registers</b>											
ASRF	C,N,OV,SZ,Z	↻	—	↻	↻	↻	↻	—	—	—	—
LSRF	C,N,OV,SZ,Z	↻	—	↻	↻	↻	↻	—	—	—	—
RLCF	C,N,SZ,Z	↻	—	↻	—	↻	↻	—	—	—	—
RLNCF	N,SZ,Z	—	—	↻	—	↻	↻	—	—	—	—
RRCF	C,N,SZ,Z	↻	—	↻	—	↻	↻	—	—	—	—
RRNCF	N,SZ,Z	—	—	↻	—	↻	↻	—	—	—	—
SLF	C,N,OV,SZ,Z	↻	—	↻	↻	↻	↻	—	—	—	—
<b>Barrel Shift Operations - W Registers (shift range -16..15)</b>											
ASRW	C,SZ,Z	↻	—	—	—	↻	↻	—	—	—	—
LSRW	C,SZ,Z	↻	—	—	—	↻	↻	—	—	—	—
MSLW	C,SZ,Z	↻	—	—	—	↻	↻	—	—	—	—
MSRW	C,SZ,Z	↻	—	—	—	↻	↻	—	—	—	—
SLW	C,SZ,Z	↻	—	—	—	↻	↻	—	—	—	—
<b>Barrel Shift Operations - Short Literals (shift range -16..15)</b>											
ASRK	C,SZ,Z	↻	—	—	—	↻	↻	—	—	—	—
LSRK	C,SZ,Z	↻	—	—	—	↻	↻	—	—	—	—
MSLK	C,SZ,Z	↻	—	—	—	↻	↻	—	—	—	—
MSRK	C,SZ,Z	↻	—	—	—	↻	↻	—	—	—	—
SLK	C,SZ,Z	↻	—	—	—	↻	↻	—	—	—	—

FOR 090-25402860



**TABLE 1-3: ROADRUNNER STATUS FLAG OPERATIONS (CONTINUED)**

PLA Mnemonic	Status Affected	C	DC	N	OV	SZ	Z	OA	OB	SA	SB
<b>DSP OPERATIONS - Accumulator Ops</b>											
ADDAB	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
ADDAC	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
LAC	None	—	—	—	—	—	—	—	—	—	—
NEGAB	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
SAC	None	—	—	—	—	—	—	—	—	—	—
SFTAC	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
SFTACK	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
SRAC	None	—	—	—	—	—	—	—	—	—	—
SUBAB	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
<b>DSP OPERATIONS - MAC Ops</b>											
CLRAC	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
ED	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
EDAC	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
MAC	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
MOVSA	None	—	—	—	—	—	—	—	—	—	—
MPY	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
MPYN	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
MSC	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
SQR	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕
SQRAC	OA,OB,SA,SB	—	—	—	—	—	—	↕	↕	↕	↕

09070457 "060401

**TABLE 1-3: ROADRUNNER STATUS FLAG OPERATIONS (CONTINUED)**

PLA Mnemonic	Status Affected	C	DC	N	OV	SZ	Z	OA	OB	SA	SB
<b>BIT OPERATIONS - W Registers</b>											
BCLR	None	—	—	—	—	—	—	—	—	—	—
BSET	None	—	—	—	—	—	—	—	—	—	—
BSW	None	—	—	—	—	—	—	—	—	—	—
BTG	None	—	—	—	—	—	—	—	—	—	—
BTST	C or Z	⇕	—	—	—	—	⇕	—	—	—	—
BTSTS	C or Z	⇕	—	—	—	—	⇕	—	—	—	—
BTSTW	C or Z	⇕	—	—	—	—	⇕	—	—	—	—
<b>BIT OPERATIONS - File Registers</b>											
BCLRF	None	—	—	—	—	—	—	—	—	—	—
BSETF	None	—	—	—	—	—	—	—	—	—	—
BTGF	None	—	—	—	—	—	—	—	—	—	—
BTSTF	Z	—	—	—	—	—	⇕	—	—	—	—
BTSTSF	Z	—	—	—	—	—	⇕	—	—	—	—
<b>BIT FIND OPERATIONS</b>											
FBCL	SZ,Z	—	—	—	—	⇕	⇕	—	—	—	—
FBCR	SZ,Z	—	—	—	—	⇕	⇕	—	—	—	—
FF0L	SZ,Z	—	—	—	—	⇕	⇕	—	—	—	—
FF0R	SZ,Z	—	—	—	—	⇕	⇕	—	—	—	—
FF1L	SZ,Z	—	—	—	—	⇕	⇕	—	—	—	—
FF1R	SZ,Z	—	—	—	—	⇕	⇕	—	—	—	—
<b>Skip OPERATIONS - W Registers</b>											
BTSC	None	—	—	—	—	—	—	—	—	—	—
BTSS	None	—	—	—	—	—	—	—	—	—	—
<b>Skip OPERATIONS - File Registers</b>											
BTFSC	None	—	—	—	—	—	—	—	—	—	—
BTFSS	None	—	—	—	—	—	—	—	—	—	—
<b>Inc/Dec Skip OPERATIONS - File Registers</b>											
DECFSNZ	None	—	—	—	—	—	—	—	—	—	—
DECFSZ	None	—	—	—	—	—	—	—	—	—	—
INCFSNZ	None	—	—	—	—	—	—	—	—	—	—
INCFSZ	None	—	—	—	—	—	—	—	—	—	—
<b>Compare OPERATIONS - W Registers</b>											
CP0	C,DC,N,OV,SZ,Z	⇕	⇕	⇕	⇕	⇕	⇕	—	—	—	—
CP1	C,DC,N,OV,SZ,Z	⇕	⇕	⇕	⇕	⇕	⇕	—	—	—	—
CP	C,DC,N,OV,SZ,Z	⇕	⇕	⇕	⇕	⇕	⇕	—	—	—	—
CPB	C,DC,N,OV,SZ,Z	⇕	⇕	⇕	⇕	⇕	⇕	—	—	—	—
<b>Compare OPERATIONS - Short Literals (literal 0...31)</b>											
CPLS	C,DC,N,OV,SZ,Z	⇕	⇕	⇕	⇕	⇕	⇕	—	—	—	—
CPBLS	C,DC,N,OV,SZ,Z	⇕	⇕	⇕	⇕	⇕	⇕	—	—	—	—
<b>Compare OPERATIONS - File Registers</b>											
CPF0	C,DC,N,OV,SZ,Z	⇕	⇕	⇕	⇕	⇕	⇕	—	—	—	—
CPF1	C,DC,N,OV,SZ,Z	⇕	⇕	⇕	⇕	⇕	⇕	—	—	—	—
CPF	C,DC,N,OV,SZ,Z	⇕	⇕	⇕	⇕	⇕	⇕	—	—	—	—
CPFB	C,DC,N,OV,SZ,Z	⇕	⇕	⇕	⇕	⇕	⇕	—	—	—	—
<b>Compare Skip OPERATIONS - File Registers</b>											
CPFSEQ	None	—	—	—	—	—	—	—	—	—	—
CPFSGT	None	—	—	—	—	—	—	—	—	—	—
CPFSLT	None	—	—	—	—	—	—	—	—	—	—
CPFSNE	None	—	—	—	—	—	—	—	—	—	—

[illegible][illegible]

**TABLE 1-4: ROADRUNNER OPCODE FIELD DESCRIPTIONS**

Field	Description
A	Accumulator selection bit: 0=ACCA; 1=ACCB
B	Byte mode selection bit: 0=word operation; 1=byte operation
D	Destination address bit: 0=result stored in Wd; 1=result stored in file register
S	Push or Pop shadows: 0=no shadows; 1=use shadows
z	Bit test destination: 0=C flag bit; 1=Z flag bit
dddd	Wd destination register select: 0000=W0; 1111=W15
ssss	Ws source register select: 0000=W0; 1111=W15
www	Wb base register select: 0000=W0; 1111=W15
ppp	Addressing mode for Ws source register (See Table 1-5)
qqq	Addressing mode for Wd destination register (See Table 1-6)
ggg	Literal offset addressing mode for Ws source register (See Table 1-7)
hhh	Literal offset addressing mode for Wd destination register (See Table 1-8)
xx	Pre-fetch X Destination (See Table 1-10)
yy	Pre-fetch Y Destination (See Table 1-13)
iii	Pre-fetch X Operation (See Table 1-9)
jjj	Pre-fetch Y Operation (See Table 1-12)
mmm	Multiplier source select (See Table 1-11)
aa	Accumulator write back mode (See Table 1-14)
rrrr	Barrel shift count
bbb	3-bit bit position select: 000=LSB; 111=MSB
bbbb	4-bit bit position select: 0000=LSB; 1111=MSB
f ffff ffff ffff	13-bit register file address (0x0000 to 0x1FFF)
ffff ffff ffff ffff	16-bit register file address (0x0000 to 0xFFFF)
k kkkk	5-bit literal field, constant data or label
kkkk kkkk	8-bit literal field, constant data or label
kk kkkk kkkk	10-bit literal field, constant data or label
kk kkkk kkkk kkkk	14-bit literal field, constant data or label
kkkk kkkk kkkk kkkk	16-bit literal field, constant data or label
n	1-bit vector select for trap instructions
nnnn nnnn nnnn nnnn	16-bit program offset field for relative branch/call instructions
nnnn nnnn nnnn nnn0 nnn nnnn	23-bit program address for goto/call instructions
xxxx xxxx xxxx xxxx	16-bit unused field (don't care)

**TABLE 1-5: ADDRESSING MODES FOR Ws SOURCE REGISTER (ADDRESS MODE 1)**

ppp	Addressing Mode	Source Operand	Instruction Operation <sup>(3)</sup>	Effective Address
000	Register Direct	Ws	Wd = Ws op Wb	EAs = W register number
001	Indirect	[Ws]	Wd = [Ws] op Wb	EAs = Ws
010	Indirect with post-decrement	[Ws]--	Wd = [Ws]-- op Wb	EAs = Ws; Ws <- (Ws - 1) <sup>(1)</sup> - or - Ws <- (Ws - 2) <sup>(2)</sup>
011	Indirect with post-increment	[Ws]++	Wd = [Ws]++ op Wb	EAs = Ws; Ws <- (Ws + 1) <sup>(1)</sup> - or - Ws <- (Ws + 2) <sup>(2)</sup>
100	Indirect with pre-decrement	[Ws--]	Wd = [Ws--] op Wb	Ws <- (Ws - 1) <sup>(1)</sup> ; - or - Ws <- (Ws - 2) <sup>(2)</sup> ; EAs = Ws
101	Indirect with pre-increment	[Ws++]	Wd = [Ws++] op Wb	Ws <- (Ws + 1) <sup>(1)</sup> ; - or - Ws <- (Ws + 2) <sup>(2)</sup> ; EAs = Ws
11k	(Specifies Slit5 Source for Short Literal Instructions)			
<b>Note 1:</b> For byte operations, add or subtract 1. <b>2:</b> For word operations, add or subtract 2. <b>3: Wd assumed to be in register direct mode (qqq=000).</b>				

**TABLE 1-6: ADDRESSING MODES FOR Wd DESTINATION REGISTER (ADDRESS MODE 2)**

qqq	Addressing Mode	Destination Operand	Instruction Operation <sup>(3)</sup>	Effective Address
000	Register Direct	Wd	Wd = Ws op Wb	EAd = W register number
001	Indirect	[Wd]	[Wd] = Ws op Wb	EAd = Wd
010	Indirect with post-decrement	[Wd]--	[Wd]-- = Ws op Wb	EAd = Wd; Wd <- (Wd - 1) <sup>(1)</sup> - or - Wd <- (Wd - 2) <sup>(2)</sup>
011	Indirect with post-increment	[Wd]++	[Wd]++ = Ws op Wb	EAd = Wd; Wd <- (Wd + 1) <sup>(1)</sup> - or - Wd <- (Wd + 2) <sup>(2)</sup>
100	Indirect with pre-decrement	[Wd--]	[Wd--] = Ws op Wb	Wd <- (Wd - 1) <sup>(1)</sup> ; - or - Wd <- (Wd - 2) <sup>(2)</sup> ; EAd = Wd
101	Indirect with pre-increment	[Wd++]	[Wd++] = Ws op Wb	Wd <- (Wd + 1) <sup>(1)</sup> ; - or - Wd <- (Wd + 2) <sup>(2)</sup> ; EAd = Wd
11x	(Unused)			
<b>Note 1:</b> For byte operations, add or subtract 1. <b>2:</b> For word operations, add or subtract 2. <b>3: Ws assumed to be in register direct mode (ppp=000).</b>				

**TABLE 1-7: OFFSET ADDRESSING MODES FOR W<sub>so</sub> SOURCE REGISTER (MODE 3)**

ggg	Addressing Mode	Source Operand	Effective Address
000	Register Direct	Wns	EA = W register number
001	Indirect	[Wns]	EA = Wns
010	Indirect with post-decrement	[Wns]--	EA = Wns; Wns <- (Wns - 1) <sup>(1)</sup> - or - Wns <- (Wns - 2) <sup>(2)</sup>
011	Indirect with post-increment	[Wns]++	EA = Wns; Wns <- (Wns + 1) <sup>(1)</sup> - or - Wns <- (Wns + 2) <sup>(2)</sup>
100	Indirect with pre-decrement	[Wns--]	Wns <- (Wns - 1) <sup>(1)</sup> ; - or - Wns <- (Wns - 2) <sup>(2)</sup> ; EA = Wns
101	Indirect with register offset	[Wns+Wb]	EA = Wns + Wb <sup>(3)</sup>
11g	Indirect with positive offset by short literal Slit5 ∈ (-16...15)	[Wns+Slit5]	EA = (Wns + gwww <sub>w</sub> ) <sup>(4)</sup> - or - EA = (Wns + 2*gwww <sub>w</sub> ) <sup>(5)</sup>
<b>Note 1:</b> For byte operations, add or subtract 1. <b>2:</b> For word operations, add or subtract 2. <b>3:</b> For byte and word operations, add 2's compliment Wb. <b>4:</b> For byte operations, add or subtract gwww <sub>w</sub> . <b>5:</b> For word operations, add or subtract (2 * gwww <sub>w</sub> ) or gwww <sub>w</sub> 0.			

060457-060404

**TABLE 1-8: OFFSET ADDRESSING MODES FOR Wnd DESTINATION REGISTER (MODE 3)**

hhh	Addressing Mode	Source Operand	Effective Address
000	Register Direct	Wnd	EA = W register number
001	Indirect	[Wnd]	EA = Wnd
010	Indirect with post-decrement	[Wnd]--	EA = Wnd; Wnd <- (Wnd - 1) <sup>(1)</sup> - or - Wnd <- (Wnd - 2) <sup>(2)</sup>
011	Indirect with post-increment	[Wnd]++	EA = Wnd; Wnd <- (Wnd + 1) <sup>(1)</sup> - or - Wnd <- (Wnd + 2) <sup>(2)</sup>
100	Indirect with pre-decrement	[Wnd--]	Wnd <- (Wnd - 1) <sup>(1)</sup> ; - or - Wnd <- (Wnd - 2) <sup>(2)</sup> ; EA = Wnd
101	Indirect with register offset	[Wnd+Wb]	EA = Wnd + Wb <sup>(3)</sup>
11h	Indirect with positive offset by short literal Slit5 ∈ (-16...15)	[Wnd+Slit5]	EA = (Wnd + hwww <sup>(4)</sup> ) - or - EA = (Wnd + 2*hwww <sup>(5)</sup> )
<b>Note 1:</b> For byte operations, add or subtract 1. <b>2:</b> For word operations, add or subtract 2. <b>3:</b> For byte and word operations, add 2's compliment Wb. <b>4:</b> For byte operations, add or subtract hwww. <b>5:</b> For word operations, add or subtract (2 * hwww) or hwww0.			

**TABLE 1-9: X DATA SPACE PREFETCH OPERATION**

iiii	Operation
0000	Wxp=[W4]
0001	Wxp=[W4], W4 = W4 + 2
0010	Wxp=[W4], W4 = W4 + 4
0011	Wxp=[W4], W4 = W4 + 6
0100	No Prefetch for X Data Space
0101	Wxp=[W4], W4 = W4 - 6
0110	Wxp=[W4], W4 = W4 - 4
0111	Wxp=[W4], W4 = W4 - 2
1000	Wxp=[W5]
1001	Wxp=[W5], W5 = W5 + 2
1010	Wxp=[W5], W5 = W5 + 4
1011	Wxp=[W5], W5 = W5 + 6
1100	Wxp=[W5+W8]
1101	Wxp=[W5], W5 = W5 - 6
1110	Wxp=[W5], W5 = W5 - 4
1111	Wxp=[W5], W5 = W5 - 2

**TABLE 1-10: X DATA SPACE PREFETCH DESTINATION**

xx	Wxp
00	W0
01	W1
10	W2
11	W3

**TABLE 1-11: MAC OR MPY SOURCE OPERANDS**

mmm	Multiplicands
000	W0 * W1
001	W0 * W2
010	W0 * W3
011	Invalid (CLRAC instruction)
100	W1 * W2
101	W1 * W3
110	W2 * W3
111	Invalid (MOVS instruction)

**TABLE 1-12: Y DATA SPACE PREFETCH OPERATION**

jjjj	Operation
0000	Wyp=[W6]
0001	Wyp=[W6], W6 = W6 + 2
0010	Wyp=[W6], W6 = W6 + 4
0011	Wyp=[W6], W6 = W6 + 6
0100	No Prefetch for Y Data Space
0101	Wyp=[W6], W6 = W6 - 6
0110	Wyp=[W6], W6 = W6 - 4
0111	Wyp=[W6], W6 = W6 - 2
1000	Wyp=[W7]
1001	Wyp=[W7], W7 = W7 + 2
1010	Wyp=[W7], W7 = W7 + 4
1011	Wyp=[W7], W7 = W7 + 6
1100	Wyp=[W7+W8]
1101	Wyp=[W7], W7 = W7 - 6
1110	Wyp=[W7], W7 = W7 - 4
1111	Wyp=[W7], W7 = W7 - 2

**TABLE 1-13: Y DATA SPACE PREFETCH DESTINATION**

yy	Wyp
00	W0
01	W1
10	W2
11	W3

**TABLE 1-14: MAC ACCUMULATOR WRITE BACK SELECTIONS**

aa	Multiplicands
00	W9 = Other Accumulator (direct)
01	[W9]++ = Other Accumulator (indirect, post-increment)
10	No write back
11	Invalid (MPYxxx instruction)